Изучаем C# через разработку игр на Unity

Пятое издание

Харрисон Ферроне

Packt›

Learning C# by Developing Games with Unity 2020 *Fifth Edition*

An enjoyable and intuitive approach
to getting started with C# programming and Unity

Harrison Ferrone

Packt

BIRMINGHAM - MUMBAI

Харрисон Ферроне

Изучаем C# через разработку игр на Unity

Пятое издание

ПИТЕР®

Санкт-Петербург • Москва • Минск

2022

ББК 32.973.2-018.1
УДК 004.43
Ф43

Ферроне Харрисон

Ф43 Изучаем C# через разработку игр на Unity. 5-е издание. — СПб.: Питер, 2022. — 400 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2932-4

Изучение C# через разработку игр на Unity — популярный способ ускоренного освоения мощного и универсального языка программирования, используемого для решения прикладных задач в широком спектре предметных областей. Эта книга дает вам возможность с нуля изучить программирование на C# без зубодробительных терминов и непонятной логики программирования, причем процесс изучения сопровождается созданием простой игры на Unity.

В пятом издании изложены последние версии всех современных функций C# на примерах из игрового движка Unity, а также добавлена новая глава о промежуточных типах коллекций. Вы начнете с основ программирования и языка C#, узнаете основные концепции программирования на C#, включая переменные, классы и объектно-ориентированное программирование. Освоив программирование на C#, переключитесь непосредственно на разработку игр на Unity и узнаете, как написать сценарий простой игры на C#. На протяжении всей книги описываются лучшие практики программирования, которые помогут вам вывести свои навыки Unity и C# на новый уровень. В результате вы сможете использовать язык C# для создания собственных реальных проектов игр на Unity.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1800207806 англ.

ISBN 978-5-4461-2932-4

Краткое содержание

Оглавление

Достигнув совершенства, люди исчезают.

Теренс Хэнбери Уайт.
Король былого и грядущего

Об авторе

Харрисон Ферроне родился в Чикаго, штат Иллинойс. Пишет техническую документацию в Microsoft, создает учебные материалы для LinkedIn Learning и Pluralsight или занимается сайтом Рэя Вендерлиха.

Харрисон — обладатель нескольких дипломов Университета Колорадо в Боулдере и Колумбийского колледжа в Чикаго. Проработав несколько лет iOS-разработчиком в небольших стартапах и даже в одной компании из списка Fortune 500, он сделал карьеру преподавателя и ни секунды не жалел об этом. За все это время он купил много книг, обзавелся несколькими кошками, поработал за границей и постоянно задавался вопросом, почему в школах не проходят «Нейроманта» Уильяма Гибсона.

Книга так и не была бы дописана без поддержки Келси, моего компаньона в этом путешествии. Благодарю также Уилбура, Мерлина, Уолтера и Иви за их добрые сердца и готовность поддержать в трудную минуту.

О научных редакторах

Эндрю Эдмондс — опытный программист, разработчик игр и преподаватель. Получил степень бакалавра компьютерных наук Уошбернского университета и является сертифицированным программистом и преподавателем по Unity. После колледжа Эндрю три года трудился разработчиком в государственном аппарате штата Канзас, а затем еще пять лет учил старшеклассников писать код и создавать видеоигры. В то время он помог многим молодым амбициозным разработчикам игр достичь больших высот, в том числе выиграть национальный чемпионат Skills USA по разработке видеоигр в 2019 году, создав в Unity игру в виртуальной реальности. Эндрю живет в Вашингтоне с женой Джессикой и дочерьми Алисой и Адой.

Адам Бжозовски — опытный разработчик игр и клиентских приложений. Разбирается в Unity, Unreal Engine, C++, Swift и Java, что помогает ему находить правильное решение для каждого проекта.

Предисловие

Unity — один из самых популярных игровых движков в мире, которым пользуются и геймдизайнеры-любители, и профессиональные разработчики AAA-проектов, и киностудии. Обычно Unity считают 3D-движком, но у него есть немало специальных функций, реализующих поддержку всего на свете, от 2D-игр и виртуальной реальности до инструментов постпродакшена и кросс-платформенной адаптации.

Что разработчикам нравится в Unity? Например, позволяющий перетаскивать панели интерфейс, встроенные функции и, конечно, самое великолепное — возможность писать собственные сценарии C# для программирования поведения и игровой механики. Опытного программиста, владеющего другими языками, нисколько не смутит необходимость выучить язык C#, а вот тех, у кого нет опыта программирования, это может отпугнуть. И тут-то вам и пригодится данная книга, в которой мы с нуля рассмотрим основы программирования и языка C#, создав попутно полноценную и веселую игру в Unity.

Для кого эта книга

Книга создана в первую очередь для тех, кто не знаком с азами программирования и языком C#. Если же вы уже что-то понимающий новичок или даже опытный программист, знающий другой язык либо тот же C#, но теперь хотите заняться разработкой игр на Unity, то также пришли по адресу.

Структура книги

В главе 1 «Знакомство со средой» мы рассмотрим процесс установки Unity, познакомимся с основными функциями редактора и посмотрим документацию по различным вопросам C# и Unity. Затем поговорим о создании сценариев C# прямо изнутри Unity и через приложение Visual Studio, в котором и будем работать с кодом далее.

В главе 2 «Основные элементы программирования» мы начнем с разбора основных понятий программирования и посмотрим, как переменные, методы и классы соотносятся с ситуациями из нашей повседневной жизни. Затем перейдем к простым методам отладки, правилам форматирования и комментирования кода, а также посмотрим, как Unity превращает сценарии C# в компоненты.

В главе 3 «Погружение в переменные, типы и методы» мы более подробно поговорим о переменных, рассмотрим типы данных C#, соглашения об именовании, модификаторы доступа и все прочее, что необходимо для создания программы. Мы также рассмотрим, как писать методы, включать параметры и эффективно использовать возвращаемые типы. В конце рассмотрим стандартные методы класса `MonoBehavior` из Unity.

В главе 4 «Поток управления и типы коллекций» мы познакомимся с реализацией принятия решений в коде с помощью операторов `if` ... `else` и `switch`. Далее перейдем к работе с массивами, списками и словарями, а затем рассмотрим операторы итерации, которые позволяют циклически перебирать коллекции. В конце главы мы рассмотрим операторы цикла с условием и специальным типом данных C# — перечислением.

В главе 5 «Работа с классами, структурами и ООП» мы впервые попробуем создавать свои классы и их экземпляры, а также структуры. Рассмотрим порядок создания конструкторов, добавление переменных и методов, основы создания подклассов и наследования. В конце обсудим концепцию объектно-ориентированного программирования в целом, а также ее применение в C#.

В главе 6 «Погружение в Unity» мы переключимся с синтаксиса C# на разработку игр, создание уровней и применение инструментов Unity.

Для начала ознакомимся с основами документации игрового дизайна, научимся фиксировать геометрию уровня, добавим освещение и простую систему частиц.

В главе 7 «Движение, управление камерой и столкновения» мы обсудим различные подходы к перемещению объекта-игрока и настройке камеры с видом от третьего лица. Для создания более реалистичного передвижения задействуем элементы физики Unity, посмотрим, как работать с коллайдерами и «ловить» взаимодействия на сцене.

В главе 8 «Программируем механику игры» мы узнаем само понятие игровой механики и то, как эффективно применять эту концепцию. Начнем с добавления простого действия — прыжка, затем реализуем механику стрельбы и, взяв уже готовый код из предыдущих глав, добавим логику для обработки сбора предметов.

В главе 9 «Основы ИИ и поведение врагов» мы начнем с краткого обсуждения реализации искусственного интеллекта в играх и тех концепций, которые понадобятся нам в игре Hero Born. Мы рассмотрим вопросы навигации в Unity, научимся пользоваться геометрией уровня и навигационной сеткой, умными агентами, а также реализуем автоматическое движение противника.

В главе 10 «Снова о типах, методах и классах» мы более подробно поговорим о типах данных, углубимся в изучение методов и дополнительных поведений, которые можно использовать для реализации более сложных классов. Вы еще больше узнаете об универсальности и широте применения языка C#.

В главе 11 «Знакомство со стеками, очередями и HashSet» мы рассмотрим более сложные типы коллекций и их особенности, поговорим об использовании стеков, очередей и типа `HashSet`, а также рассмотрим ситуации, для которых каждый из этих типов подходит просто идеально.

В главе 12 «Обобщения, делегаты и многое другое» подробно описаны более сложные функции языка C# и их применение в реальных практических задачах. Начнем с концепции обобщенного программирования, затем рассмотрим понятия делегирования, событий и обработку исклю-

чений. В конце главы вкратце изучим общие паттерны проектирования и подготовим фундамент для вашего дальнейшего путешествия в мир разработки.

В главе 13 «Путешествие продолжается» мы вспомним все то, что изучили в данной книге, и рассмотрим примеры ресурсов для дальнейшего изучения C# и Unity. Это будут и материалы в Интернете, и информация о сертификации, и множество моих любимых каналов с видеоуроками.

Как получить от книги максимум

Чтобы получить максимум пользы из данного путешествия в страну C# и Unity, вам понадобится всего ничего: любопытство и желание учиться. Это значит, вам стоит выполнить все задания рубрик «Время действовать», «Испытание героя» и «Контрольные вопросы», поскольку вам потребуется закрепить вновь обретенные знания. Наконец, прежде, чем двигаться дальше, всегда полезно возвращаться к пройденным темам и главам, чтобы освежить или закрепить понимание материала. На неустойчивом фундаменте дом не построить.

Кроме того, вам нужно будет установить на компьютер актуальную версию Unity. Рекомендуется версия 2020 или более поздняя. Все примеры кода были протестированы на Unity 2020.1 и должны без проблем работать и в более новых версиях.

Программное обеспечение, используемое в книге:

- Unity 2020.1 или новее;
- Visual Studio 2019 или новее;
- C# 8.0 или новее.

Прежде чем начать, убедитесь, что ваш компьютер соответствует системным требованиям Unity: docs.unity3d.com/ru/current/Manual/system-requirements.html. Это требования для Unity 2019, но они актуальны и для более новых версий.

Скачивание файлов с примерами кода

Файлы с примерами кода для этой книги можно скачать на сайте github.com.

Чтобы скачать файлы кода, выполните следующие действия.

1. Перейдите по ссылке github.com/PacktPublishing/Learning-C-8-by-Developing-Games-with-Unity-2020.
2. Нажмите кнопку ↓ Code и выберите пункт Download ZIP.
3. Скачайте ZIP-архив с примерами кода.

После загрузки файла вам нужно будет распаковать его с помощью последней версии следующего ПО:

- WinRAR/7-Zip для Windows;
- Zipeg/iZip/UnRarX для macOS;
- 7-Zip/PeaZip для Linux.

Скачивание цветных изображений

Вы также можете скачать PDF-файл с цветными снимками экрана и графиками, приведенными в оригинальной книге, по ссылке: static.packt-cdn.com/downloads/9781800207806_ColorImages.pdf.

Условные обозначения

В этой книге используются следующие обозначения.

`Моноширинным` шрифтом оформляются фрагменты кода в тексте, имена таблиц баз данных, имена файлов и путей, расширения файлов, вводимые пользователем данные и имена пользователей Twitter. Например, «Рассмотрим сценарий `LearningCurve`».

Блок кода форматируется следующим образом:

```
public string firstName = "Harrison";
```

Если необходимо заострить внимание на определенной части кода, то эти строки или элементы выделяются **полужирным моноширинным** шрифтом:

```
accessModifier returnType UniqueName(parameterType parameterName) {
    method body
}
```

Курсивом оформляются новые термины.

Шрифтом без засечек оформляются URL, имена папок, важное слово или слова, которые вы видите на экране. Например, это могут быть пункты меню, фразы в диалоговых окнах: «Выберите команду Create ▸ 3D Object ▸ Capsule на панели Hierarchy».

Предупреждения или важные примечания оформлены так.

Подсказки и полезные «фишки» оформлены так.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1 Знакомство со средой

В массовой культуре программисты часто представляются эдакими аутсайдерами, одиночками, хакерами до мозга костей, которые имеют фантастические способности к алгоритмическому мышлению, странные анархические наклонности и притом совершенно не адаптированы к социальной жизни. Конечно, это не так, но стоит отметить, что изучение программирования коренным образом меняет ваш взгляд на мир. Хорошо здесь то, что ваш любознательный ум быстро адаптируется к этому новому образу мышления и даже начинает получать от него удовольствие.

Даже в повседневной жизни мы используем аналитические навыки, которые можно перенести в программирование, и зачастую вам просто не хватает подходящего языка и синтаксиса, который позволил бы показать эти навыки в коде. Ну, к примеру, вы знаете свой возраст? Это переменная. Когда вы переходите улицу, вы, скорее всего, как и все, смотрите на дорогу в обоих направлениях, а затем начинаете движение. В жизни это оценка различных условий, а в программировании — управление потоком выполнения. Глядя на банку с консервами, вы подсознательно определяете, что у нее есть какие-то свойства: форма, вес, содержимое. Это объект класса! Ну вы поняли.

Имея опыт реальной жизни, вы можете перенести его в мир программирования. Вам необходимо знать, как настроить среду разработки, работать с приложениями и где искать помощи, когда она потребуется. Поэтому в начале нашего приключения в страну C# мы рассмотрим следующие вопросы:

- начало работы в Unity;
- работу в Visual Studio;
- использование C# в Unity;
- изучение документации.

Технические требования

Когда необходимо нечто описать, иногда проще начать с того, чем оно не является. Так вот, в этой книге мы не ставим цель охватить все тонкости игрового движка Unity и разработки игр. Мы начнем с постепенного рассмотрения необходимых тем на базовом уровне, а затем более подробно продолжим в главе 6. При этом все это делается лишь для того, чтобы начать изучать язык программирования C# с нуля в увлекательной и доступной манере.

Данная книга предназначена для начинающих программистов, поэтому если у вас нет опыта работы с C# или Unity, то вы по адресу! Если же немного работали с редактором Unity, но не занимались программированием, то... все равно пришли куда надо. И даже если вы немного знакомы и с C#, и с Unity, но хотите изучить их глубже и рассмотреть более продвинутые темы, в более поздних главах книги вы найдете искомое.

Если у вас уже есть серьезный опыт программирования на других языках, то можете смело пропустить теорию для начинающих и сразу переходить к тому, что вас интересует. А можете никуда не спешить и освежить в памяти основы.

Начинаем работу в Unity 2020

Если у вас еще не установлена среда Unity или ваша версия устарела, то вам придется немного поработать. Выполните следующие действия.

1. Зайдите на сайт https://www.unity.com (вид сайта показан на рис. 1.1).
2. Нажмите кнопку Get Started, чтобы перейти на страницу магазина Unity.

Вид главной страницы может отличаться от приведенного на снимке экрана. В этом случае вы можете перейти по прямой ссылке store.unity.com.

Не переживайте — вы можете получить Unity совершенно бесплатно!

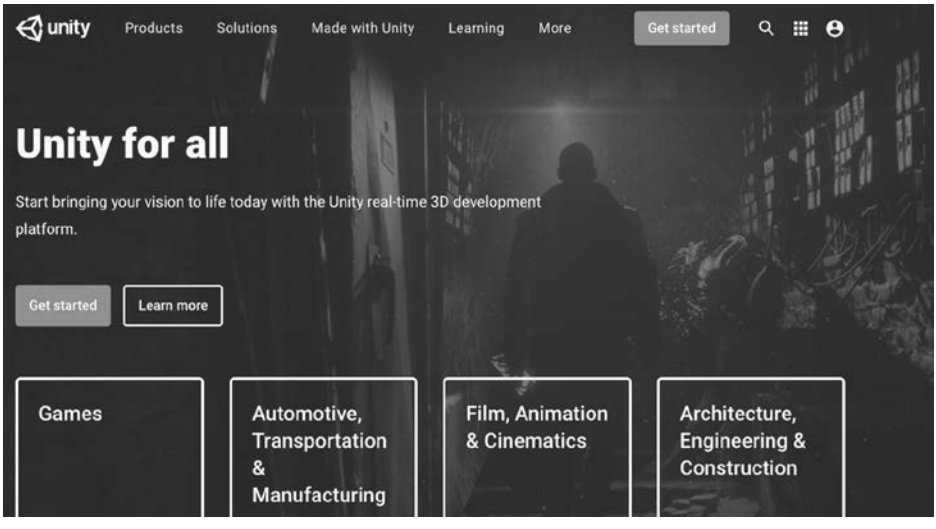


Рис. 1.1

3. Перейдите на вкладку Individual и выберите слева параметр Personal. В других, платных конфигурациях есть более продвинутые функции и подписки на сервисы (рис. 1.2). При желании можете почитать о них самостоятельно.

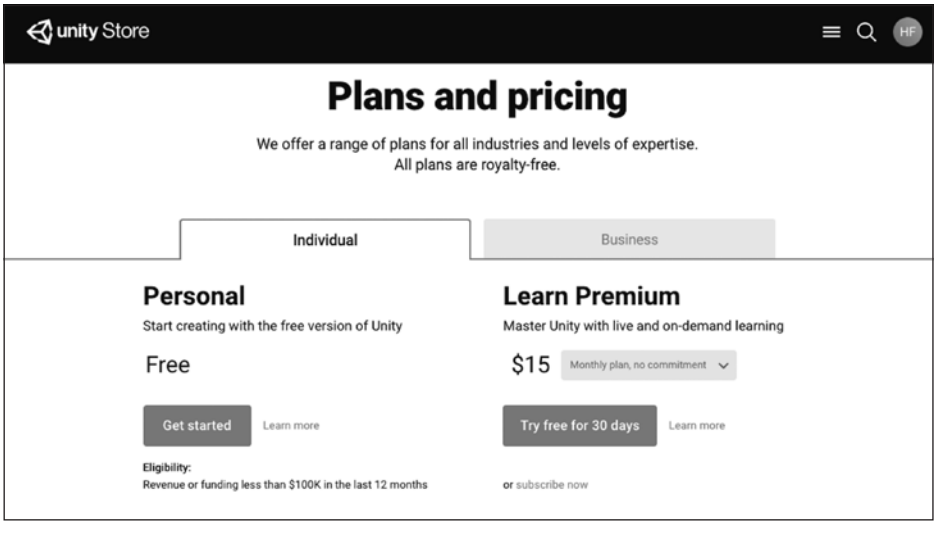


Рис. 1.2

После того как вы выберете тариф, вас спросят, впервые ли вы скачиваете Unity или уже занимались им ранее.

4. Нажмите кнопку Start here в разделе First-time Users (рис. 1.3).

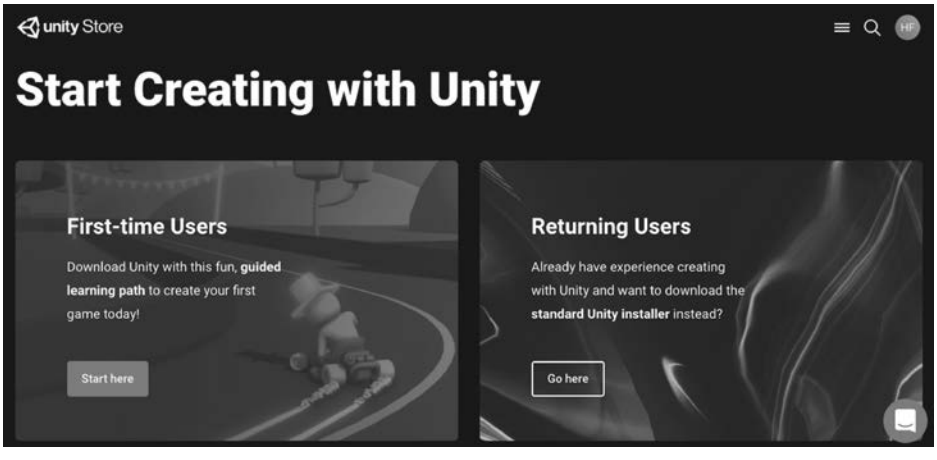


Рис. 1.3

5. Нажмите кнопку Agree and download, чтобы скачать Unity Hub, как показано на рис. 1.4.

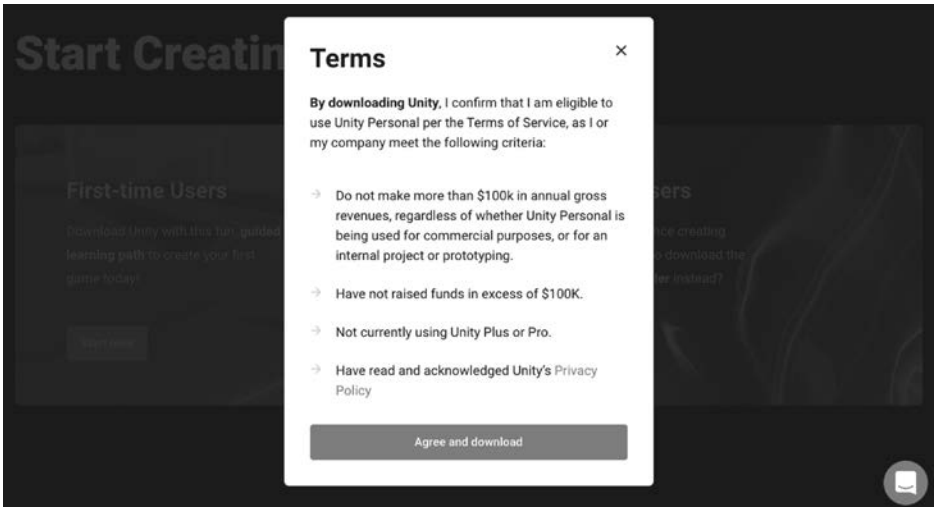


Рис. 1.4

После того как загрузка будет завершена, выполните следующие действия.

1. Откройте загруженный файл (двойным щелчком кнопкой мыши).
2. Примите пользовательское соглашение.
3. Следуйте инструкциям по установке. Когда все получится, запустите приложение Unity Hub, и вы увидите следующий экран (рис. 1.5).

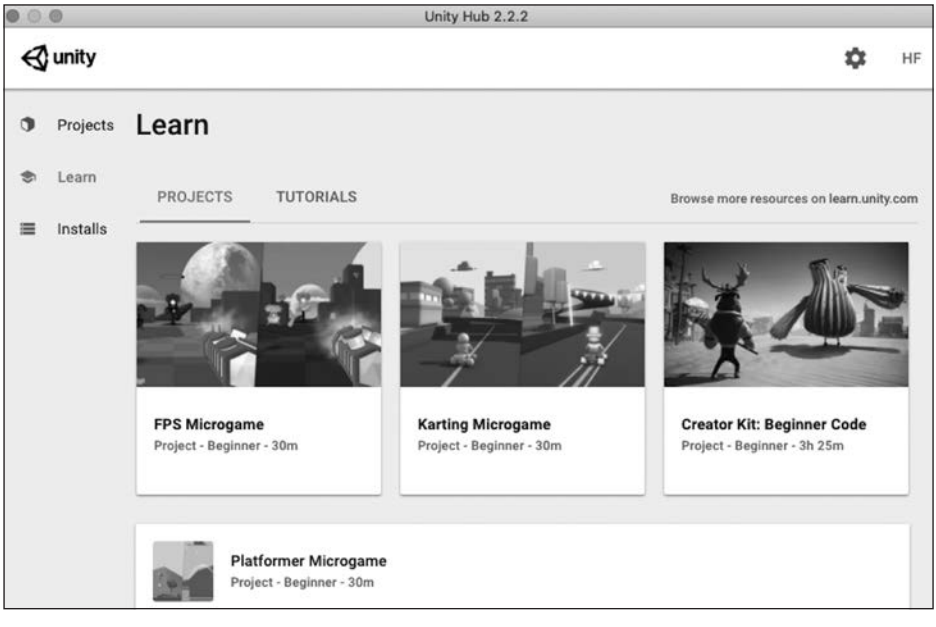


Рис. 1.5

В самой новой версии Unity Hub при первом открытии приложения запускается специальный мастер начала работы. Вы можете проследовать его указаниям, если хотите. Далее мы рассмотрим, как создать новый проект, не прибегая к мастеру, поскольку он срабатывает только при первом запуске.

4. Запустите Unity Hub, перейдите на вкладку Installs в меню слева и нажмите кнопку ADD (рис. 1.6).

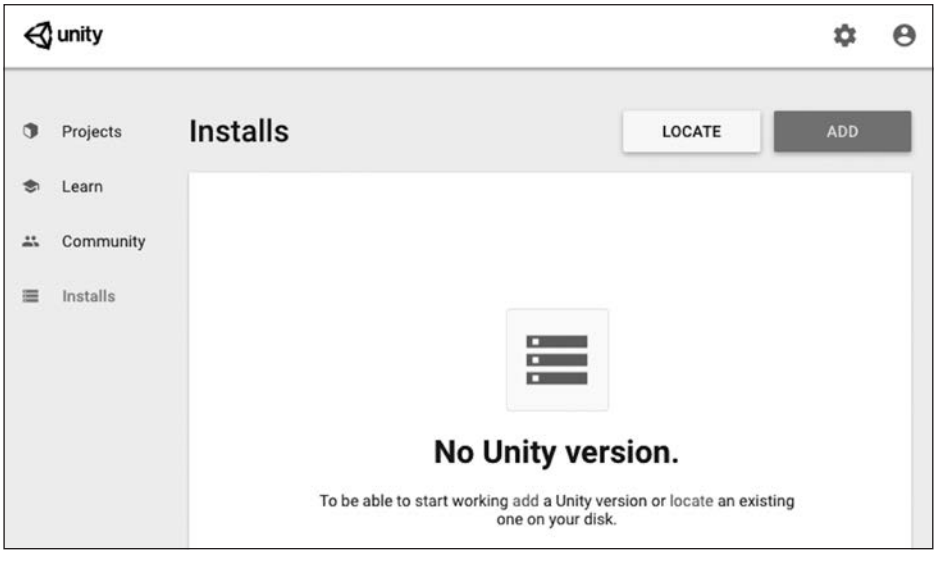


Рис. 1.6

На момент написания книги новейшая версия Unity 2020 еще находится в стадии альфа-тестирования, но вы можете выбрать эту версию из списка Latest Official Releases (рис. 1.7).

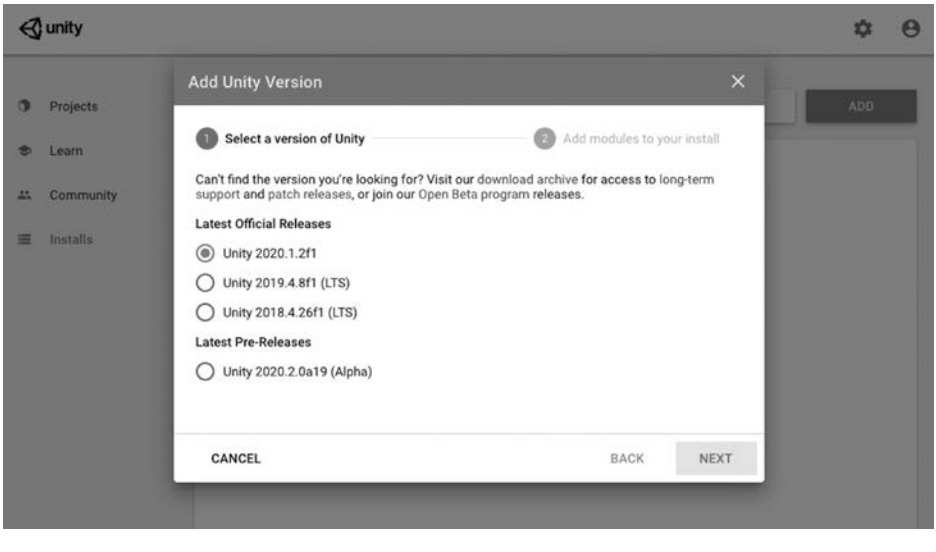


Рис. 1.7

Для работы с примерами из книги вам не понадобятся какие-либо модули для конкретных платформ, поэтому данный шаг можно пропустить. Если вы захотите загрузить их позже, то всегда можете нажать кнопку More (значок с тремя точками) в правом верхнем углу любой версии в окне Installs (рис. 1.8).

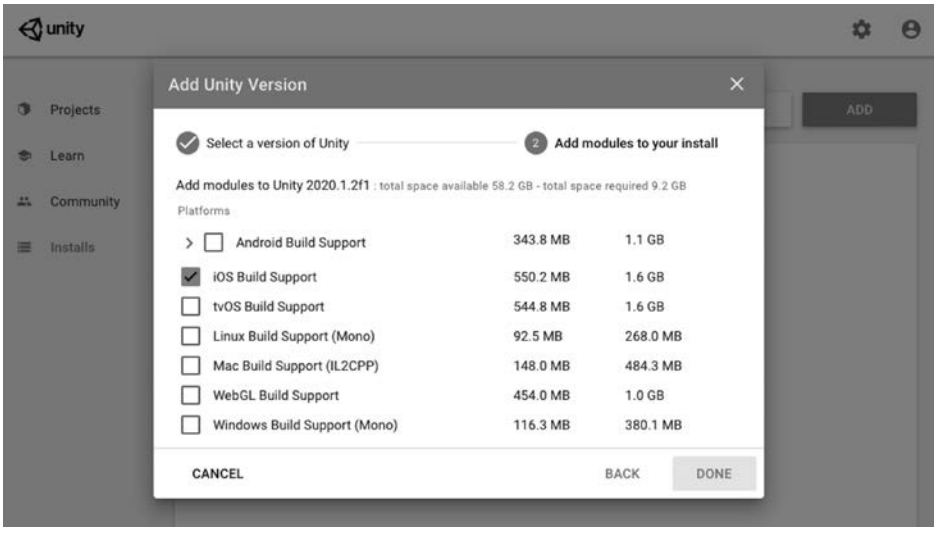


Рис. 1.8

Когда установка будет завершена, вы увидите новую версию программы на панели Installs (рис. 1.9).

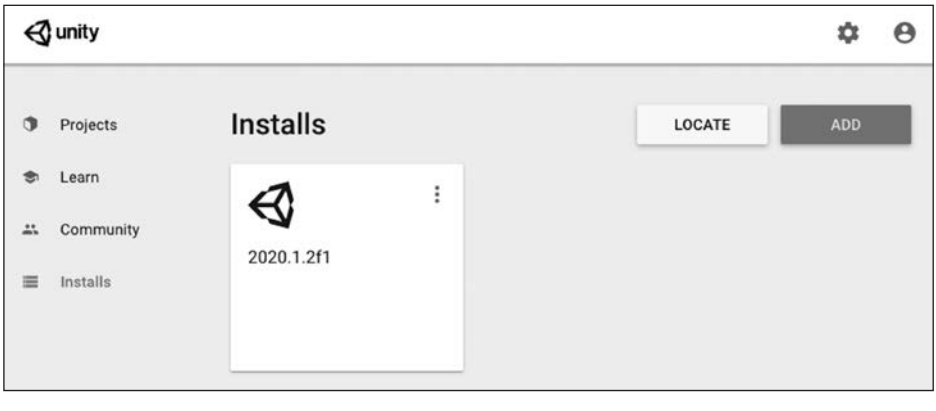


Рис. 1.9

Дополнительную информацию и ресурсы приложения Unity Hub можно найти по адресу docs.unity3d.com/ru/current/Manual/GettingStartedInstallingHub.html.

Всегда что-то может пойти не так, поэтому обязательно ознакомьтесь со следующим подразделом, если вы работаете в macOS Catalina или более новой версии.

Если у вас macOS

Если вы работаете на Mac под управлением операционной системы Catalina или более поздней версии, то в этой ОС есть известная проблема использования Unity Hub 2.2.2 (и более ранних версий) для установки Unity по описанному выше алгоритму. При наличии данной проблемы сделайте глубокий вдох, зайдите в архив загрузок Unity и скачайте нужную вам версию (https://unity3d.com/get-unity/download/archive). Не забудьте выбрать опцию Downloads (Mac) вместо Unity Hub (рис. 1.10).

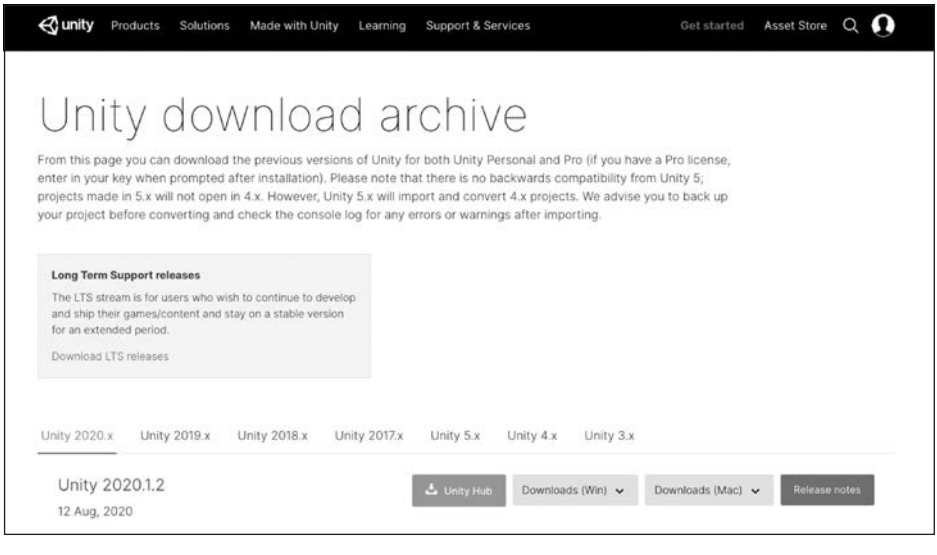


Рис. 1.10

Если вы сталкиваетесь с подобными проблемами и в операционной системе Windows, то этот алгоритм подойдет и вам.

Мы скачали установщик приложения в виде файла `.dmg`. Откройте его, следуйте инструкциям, и все будет готово в считаные мгновения (рис. 1.11)!

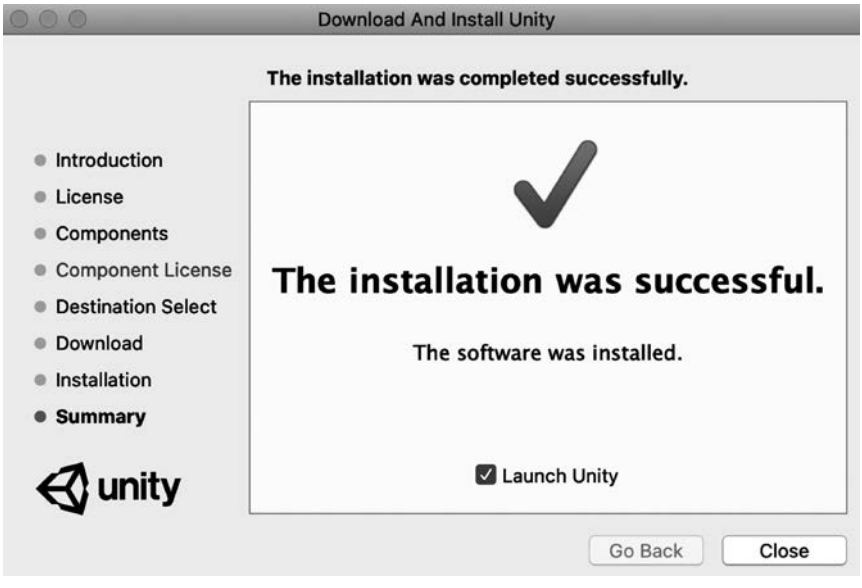


Рис. 1.11

Все примеры и снимки экрана для данной книги были созданы в Unity 2020.1.0a20. Если вы используете более новую версию, то редактор Unity может выглядеть немного иначе, но проблем от этого быть не должно.

Теперь, когда мы установили Unity Hub и Unity 2020, пора создать новый проект!

Создание нового проекта

Чтобы создать новый проект, запустите приложение Unity Hub. Если у вас есть учетная запись Unity, то войдите в систему. Если нет, то можете создать ее или нажать кнопку Skip в нижней части экрана.

Теперь настроим новый проект, щелкнув на треугольнике рядом с вкладкой NEW в правом верхнем углу (рис. 1.12).

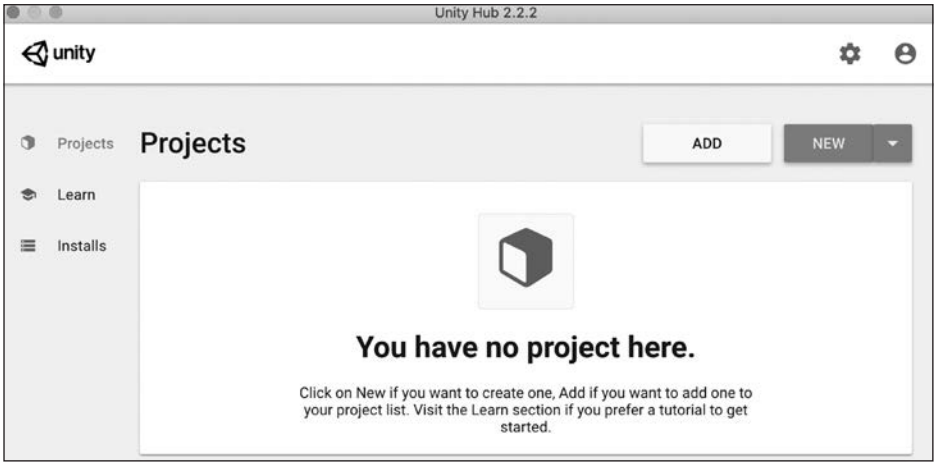


Рис. 1.12

Выберите вашу версию Unity и заполните следующие поля.

- Project name: я назову проект `Hero Born`.
- Location: здесь мы укажем, где хотим сохранить проект.
- Template: по умолчанию создается 3D-проект, поэтому можно сразу нажимать кнопку CREATE (рис. 1.13).

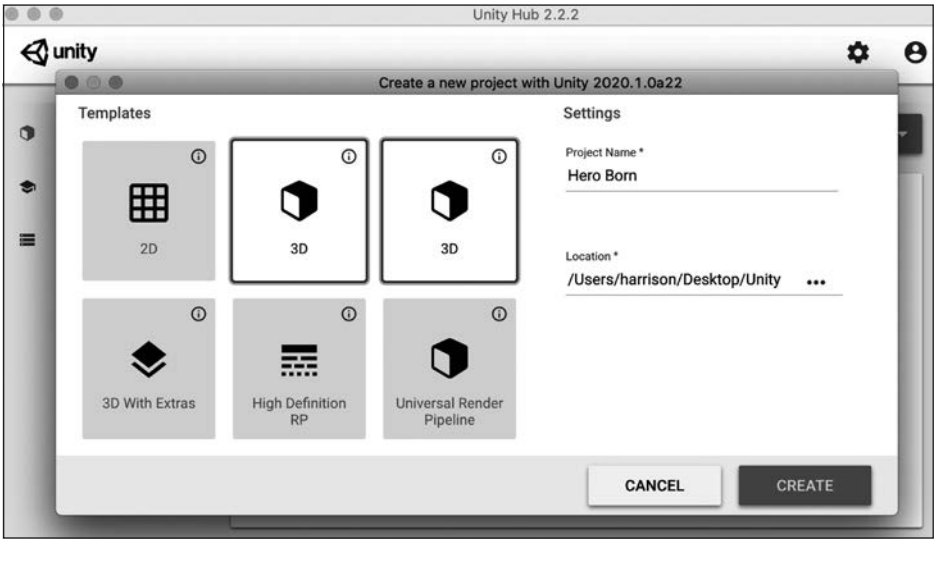


Рис. 1.13

Когда проект создан, можно переходить к изучению интерфейса Unity.

Интерфейс программы

Когда новый проект инициализируется, перед вами появится великолепный редактор Unity! Я обозначил важные вкладки (или панели, если вам так больше нравится) на рис. 1.14.

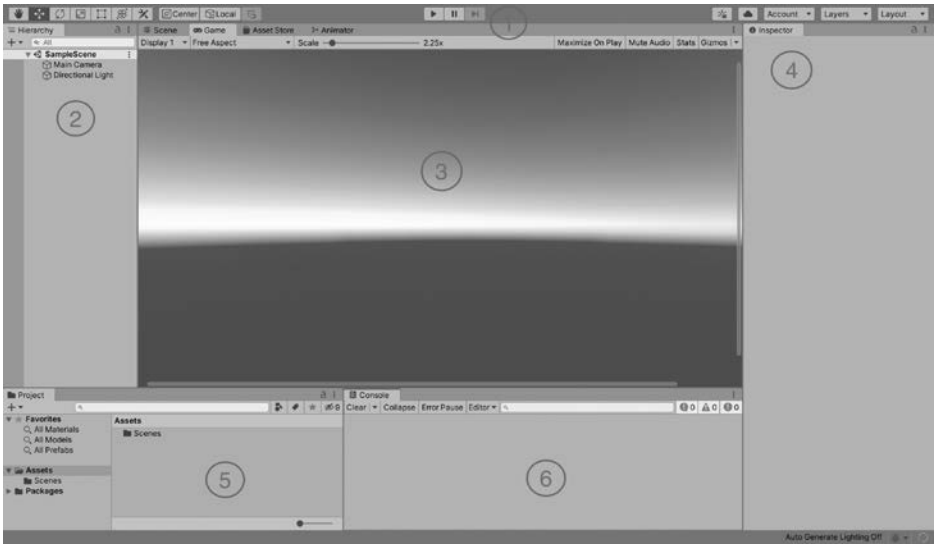


Рис. 1.14

Рассмотрим каждую из панелей более подробно.

1. Панель Toolbar — самая верхняя часть редактора Unity. На этой панели можно управлять объектами (кнопки слева), а также запускать и приостанавливать игру (центральные кнопки). Справа расположены кнопки сервисов Unity, слои-маски и опции макетов, которые мы не будем использовать в этой книге.
2. На панели Hierarchy отображаются все элементы, находящиеся в данный момент на игровой сцене. Во вновь созданном проекте есть камера и направленный источник света, но по мере создания игры эта панель начнет наполняться объектами.

3. Панели Game и Scene — самые визуально наглядные части редактора. Панель Scene — ваша рабочая поверхность, на которой вы можете размещать и перемещать 2D- и 3D-объекты. Когда вы нажимаете кнопку Play, появляется окно Game, в котором будет показана та же сцена, но уже с рабочими запрограммированными взаимодействиями.
4. Панель Inspector — обобщенный инструмент для просмотра и редактирования свойств ваших объектов. Выбрав компонент `Main Camera`, вы увидите, что у него есть несколько частей (в Unity они называются компонентами) и все они находятся именно здесь.
5. В окне Project представлены все ресурсы, которые в настоящее время находятся в вашем проекте. Если проще, то это все файлы и папки вашего проекта.
6. На панели Console станут появляться выходные данные, которые будут выводить наши сценарии. С этого момента, если мы станем говорить о выводе консоли или отладки, то будем иметь в виду именно это место.

Более подробное описание функциональности каждой панели можно найти в документации Unity по адресу docs.unity3d.com/ru/current/Manual/UsingTheEditor.html.

Понимаю, что для новичка в Unity информации с избытком, однако не беспокойтесь, ведь в дальнейшем во всех инструкциях будут подробно расписаны все необходимые шаги. Вам не придется думать и гадать, какую кнопку нажать. Поскольку с панелями разобрались, теперь приступим к созданию реальных сценариев на C#.

Использование C# с Unity

Забегая вперед, отмечу, что нужно рассматривать Unity и C# как симбиотические сущности. Unity — это движок, в котором вы будете создавать сценарии и в дальнейшем запускать их, но фактическое программирование происходит в другой программе под названием Visual Studio. Однако прямо сейчас об этом задумываться не нужно, ведь мы вернемся к данному вопросу через мгновение.

Работа со сценариями C#

Хоть мы еще и не рассмотрели основные концепции программирования, нам негде будет их применять, пока мы не узнаем, как создать сценарий C# в Unity.

Есть несколько способов создания сценариев C# из редактора.

- Выберите команду Assets ▸ Create ▸ C# Script.
- На панели Project выберите команду Create ▸ C# Script.
- Щелкните правой кнопкой мыши на панели Project (справа) и выберите команду Create ▸ C# Script из всплывающего меню.
- Выберите объект `GameObject` на панели Hierarchy и нажмите Add Component ▸ New Script.

В дальнейшем, когда нужно будет создать сценарий C#, вы можете использовать любой из этих способов, который вам больше нравится.

Эти же способы позволяют создавать и другие ресурсы и объекты, а не только сценарии C#. Я не буду перечислять все их каждый раз, когда создаем что-то новое, поэтому запомните их сразу и надолго.

Из соображений сохранения порядка мы будем хранить наши ресурсы и сценарии в специально отведенных для них папках. Это нужно делать не только при работе с Unity, и ваши коллеги будут вам благодарны (обещаю).

1. Выберите команду Create ▸ Folder (или любой другой способ, который вам больше нравится) на панели Project и назовите папку Scripts (рис. 1.15).
2. Дважды щелкните на папке Scripts и создайте новый сценарий C#. По умолчанию сценарий будет называться `NewBehaviourScript`, но имя файла будет выделено, поэтому вы тут же сможете переименовать его. Введите название `LearningCurve` и нажмите клавишу Enter (рис. 1.16).

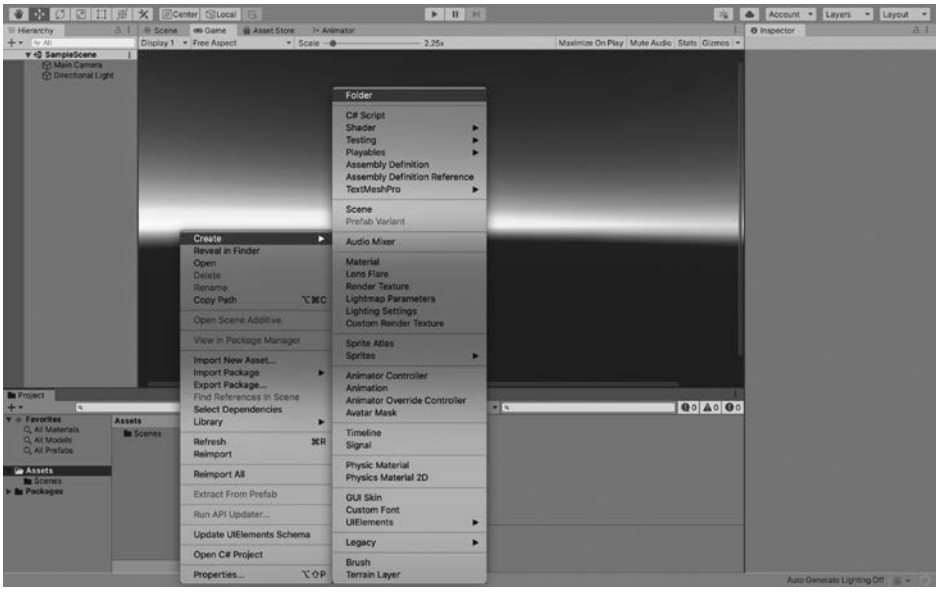


Рис. 1.15

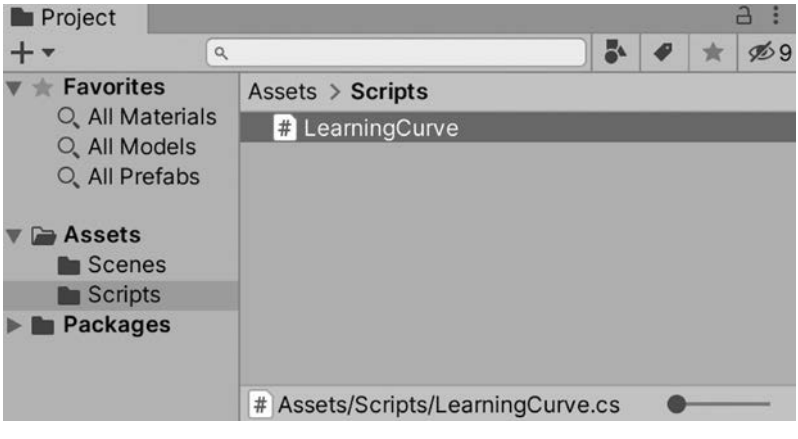


Рис. 1.16

Мы только что создали подпапку с именем Scripts, и она показана на рис. 1.16. Внутри этой папки мы создали сценарий C# с именем `LearningCurve.cs` (тип файла `.cs` означает *C-Sharp*, если вдруг интересно), и он стал одним из активов проекта Hero Born. Нам осталось лишь открыть файл в Visual Studio!

Редактор Visual Studio

В самом Unity можно создавать и хранить сценарии C#, но редактировать их придется с помощью Visual Studio. Дистрибутив Visual Studio идет в комплекте с Unity и откроется автоматически, если вы дважды щелкнете на любом сценарии C# из редактора.

Время действовать. Открываем файл C#

Unity синхронизируется с Visual Studio при первом открытии файла. Самый простой способ сделать это — выбрать сценарий на панели Project.

Дважды щелкните на файле `LearningCurve.cs` (рис. 1.17).

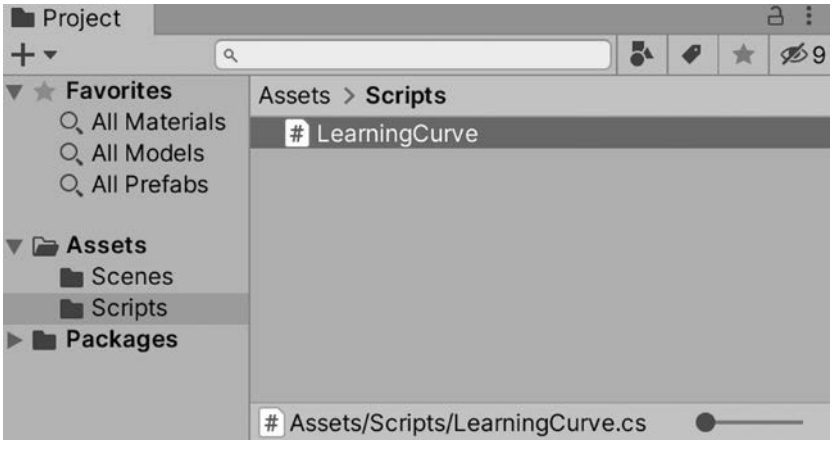


Рис. 1.17

В результате файл C# откроется в Visual Studio (рис. 1.18).

Если файл по умолчанию открывается в другом приложении, то выполните следующие действия.

1. Выберите команду Unity ▸ Preferences в верхнем меню, а затем на левой панели выберите External Tools.
2. Измените параметр External Script Editor на Visual Studio, как показано на рис. 1.19.

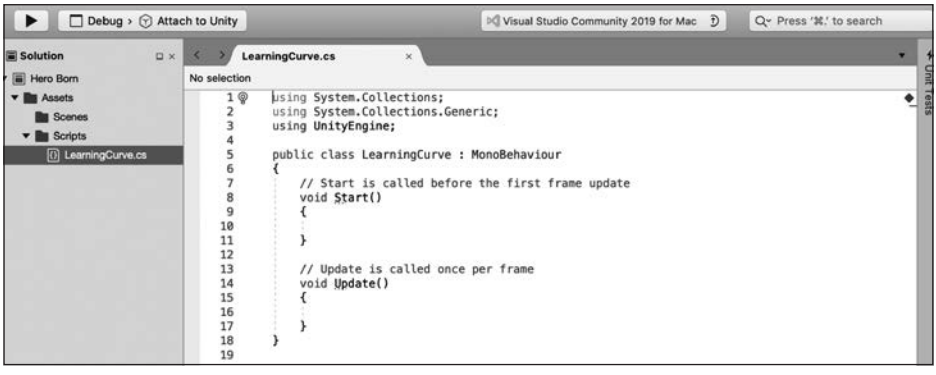


Рис. 1.18

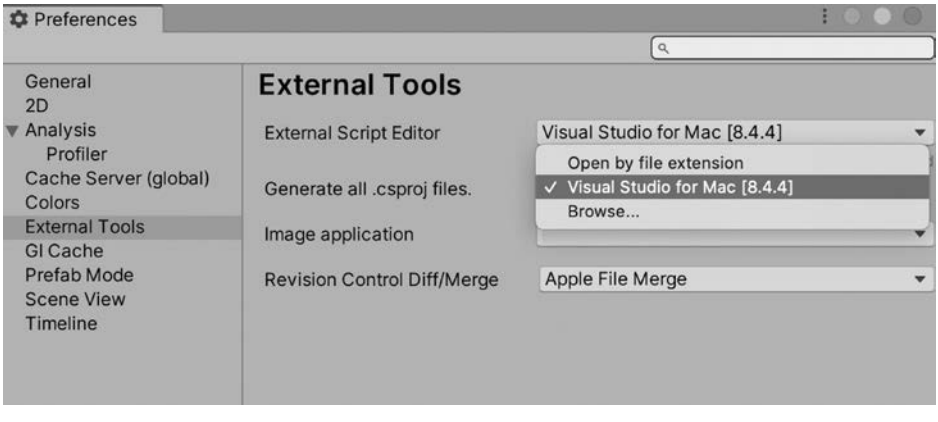


Рис. 1.19

В левой части интерфейса Visual Studio вы увидите структуру папок, отражающую структуру в Unity, и работать с ней можно будет аналогичным образом. Справа находится сам редактор кода, в котором и творится волшебство. В приложении Visual Studio гораздо больше возможностей, но для начала нам будет достаточно и этого.

Интерфейс Visual Studio в Windows и macOS отличается, но код, который мы будем использовать в данной книге, будет одинаково правильно работать в обеих ОС. Все снимки экрана для книги были сделаны в среде Mac, поэтому, если на вашем компьютере все выглядит по-другому, волноваться не надо.

Остерегайтесь несоответствий в именах

Одна из распространенных ошибок, которая сбивает с толку начинающих программистов, — имена файлов, а точнее, несоответствие этих имен. Проиллюстрируем проблему с помощью строки 5 из скриншота файла C# в Visual Studio, показанного выше, на рис. 1.18:

```
public class LearningCurve : MonoBehaviour
```

Имя класса `LearningCurve` совпадает с именем файла `LearningCurve.cs`. **Это обязательное требование.** Пусть даже вы еще и не знаете, что такое класс, важно помнить: в Unity имя файла и имя класса должны совпадать. Но если вы используете C# вне Unity, то это правило соблюдать не обязательно.

Когда вы создаете файл сценария C# в Unity, имя файла на панели Project сразу находится в режиме редактирования и готово к изменению. Лучше взять за привычку сразу переименовывать файл. Если вы переименуете сценарий позже, то имя файла и имя класса не будут совпадать. Тогда имя файла изменится, но пятая строка будет выглядеть следующим образом:

```
public class NewBehaviourScript : MonoBehaviour
```

Если вы случайно сделаете так, то это не вызовет катастрофу. Чтобы все починить, достаточно будет войти в Visual Studio и заменить имя класса `NewBehaviourScript` на имя вашего сценария C#.

Синхронизация файлов C#

В рамках симбиотических отношений Unity и Visual Studio поддерживают связь друг с другом, чтобы синхронизировать контент. Это значит, что если вы добавляете, удаляете или изменяете файл сценария в одном приложении, то другое приложение автоматически увидит эти изменения.

А что произойдет, если сработает закон Мёрфи и синхронизация попросту не выполнится? Столкнувшись с такой ситуацией, сделайте глубокий вдох, найдите проблемный сценарий, щелкните правой кнопкой мыши и выберите команду Refresh.

Теперь вы знаете основы создания сценариев, поэтому пришло время поговорить о поиске и эффективном использовании полезных ресурсов.

Изучение документации

Последняя тема, которую мы осветим в нашем первом контакте с Unity и C#, — документация. Это скучно, знаю, но важно как можно раньше сформировать правильные навыки работы с новыми языками программирования или средами разработки.

Доступ к документации Unity

Как только вы начнете писать серьезные сценарии, вам придется довольно часто работать с документацией Unity, поэтому будет полезно как можно раньше узнать, где ее найти. В руководстве по программированию (Reference Manual) можно почитать о конкретном компоненте или теме, а примеры кода можно найти в Scripting Reference (справочнике по сценариям).

Время действовать. Читаем руководство по программированию

У каждого объекта `GameObject` (элемент на панели Hierarchy) на сцене есть компонент `Transform`, который определяет его положение, поворот и масштаб. Для простоты найдем в руководстве по программированию компонент `Transform` у камеры.

1. На панели Hierarchy выберите объект `Main Camera`.
2. Перейдите на вкладку `Inspector` и щелкните на значке информации (знак вопроса) в правом верхнем углу компонента `Transform` (рис. 1.20).

После этого откроется браузер, а в нем появится руководство по программированию на странице Transform. Все компоненты Unity снабжены справкой, поэтому, если вам потребуется узнать больше об их работе, вы знаете, что делать (рис. 1.21).

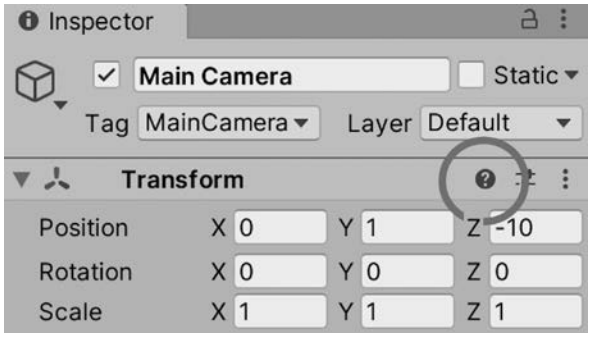


Рис. 1.20

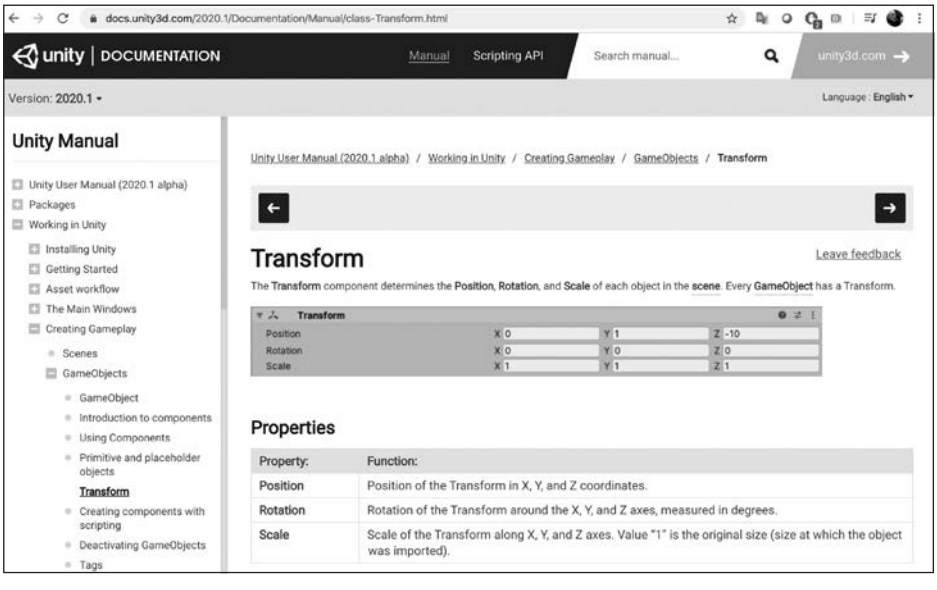


Рис. 1.21

Время действовать. Читаем Scripting Reference

Итак, у нас открыто руководство по программированию, но что, если нам нужны конкретные примеры кода, связанные с компонентом `Transform`? Тут тоже все просто — нам поможет Scripting Reference.

Выберите Scripting API или ссылку Switch to Scripting под именем компонента или класса (в данном случае компонента `Transform`).

После этого руководство по программированию автоматически переключится на Scripting Reference для компонента `Transform` (рис. 1.22).

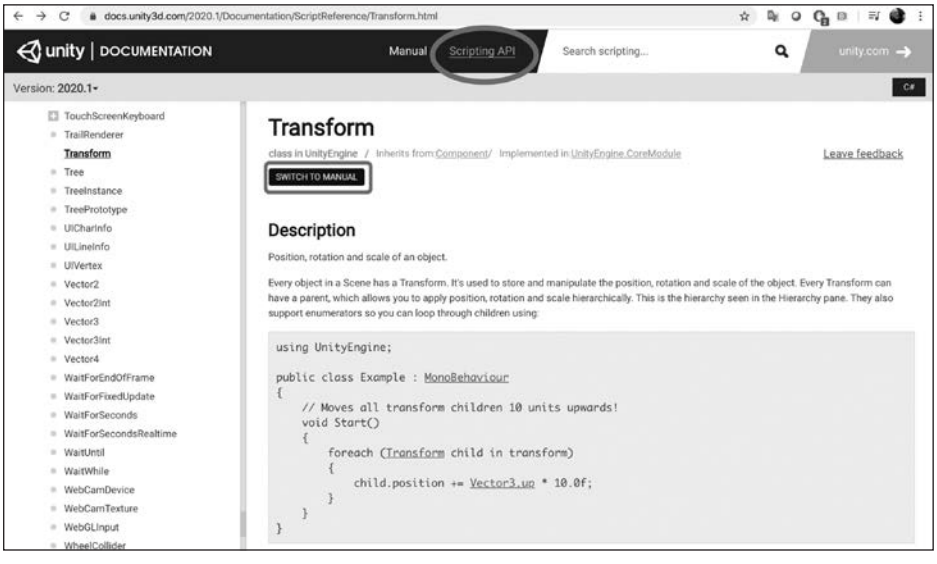


Рис. 1.22

Scripting Reference — большой документ, но таким он и должен быть. Однако это не означает, что вам нужно выучить его целиком или даже прочитать всю приведенную в нем информацию, чтобы начать писать сценарии. Это справочник, а не тест.

Если вы запутались в документации или просто не знаете, где что искать, за решением многих проблем можно обратиться к огромному сообществу разработчиков Unity:

- forum.unity.com/;
- answers.unity.com/index.html;
- discord.com/invite/unity.

Кроме того, вам нужно знать, где найти ресурсы по любому вопросу, связанному с языком C#, о чем мы и поговорим далее.

Поиск ресурсов C#

Теперь, когда у нас есть нужные ресурсы по Unity, почитаем документацию по Microsoft C# на docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/.

Существует и множество других ресурсов по языку C#, начиная от справочников и руководств по быстрому запуску и заканчивая спецификациями версий (если вам такое нравится). Все это можно найти по адресу docs.microsoft.com/ru-ru/dotnet/csharp/.

Время действовать. Находим класс C#

Откроем руководство по программированию и найдем класс `String` языка C#. Выполните одно из следующих действий.

- Введите Strings в строку поиска в верхнем левом углу веб-страницы.
- Прокрутите вниз до раздела Language Sections и щелкните на ссылке Strings (рис. 1.23).

Рис. 1.23

На странице описания класса вы увидите нечто наподобие этого (рис. 1.24). В отличие от документации по Unity справка по C# и ин-

формация о сценариях объединены в одно целое, для простоты справа вынесен список подтем. Пользуйтесь на здоровье.

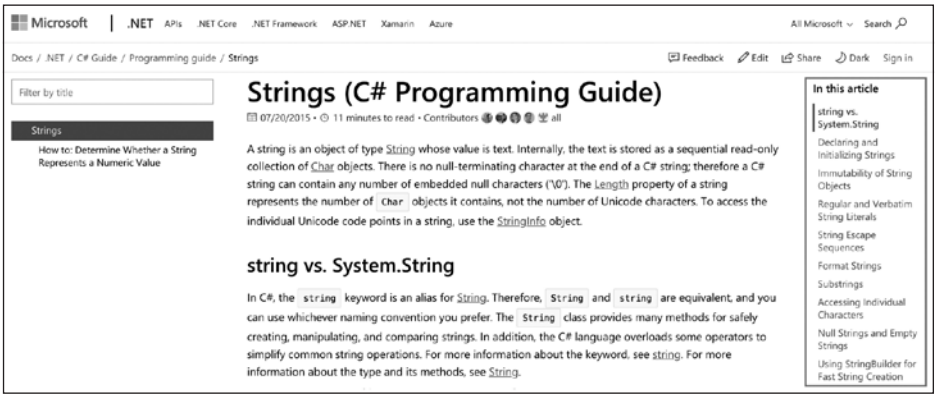


Рис. 1.24

Невероятно важно знать, куда бежать за помощью, если вы застряли или не можете решить какой-то вопрос, поэтому обязательно возвращайтесь к данному разделу, когда сталкиваетесь с проблемой.

Подведем итоги

В данной главе было довольно много информации из разряда «это здесь, а то там», и потому вам уже наверняка не терпится написать какой-нибудь код. Запуск нового проекта, создание папок и сценариев, а также поиск документации — важные вопросы, о которых легко забыть на волне ажиотажа, вызванного предстоящим приключением. Просто помните, что в текущей главе мы рассмотрели много ресурсов, которые могут вам понадобиться далее по мере чтения книги, поэтому не стесняйтесь возвращаться сюда и обращаться к этим ресурсам. Программистское мышление словно мышца, и чем больше вы над ним работаете, тем сильнее оно становится.

В следующей главе мы начнем рассматривать теорию, термины и основные концепции, которые должны будут подготовить ваш разум к программированию. Несмотря на то что материал будет в основном

понятийного характера, мы все же напишем первые строки сценария `LearningCurve`. Пристегните ремни!

Контрольные вопросы. Работа со сценариями

1. Как взаимосвязаны Unity и Visual Studio?
2. В Scripting Reference приведен пример кода, в котором показано применение определенного компонента или функции Unity. А где можно найти более подробную (не связанную с кодом) информацию о компонентах Unity?
3. Scripting Reference — большой документ. Какую его часть вам придется запомнить, прежде чем вы сможете написать сценарий?
4. В какой момент лучше всего давать имя сценарию C#?

2 Основные элементы программирования

Любой язык программирования несведущему человеку кажется китайской грамотой, и C# в этом смысле не исключение. Но не все так плохо — за непонятными словами у всех языков программирования скрываются одни и те же основные элеметы. Переменные, методы и классы (или объекты) — вот в чем заключается суть традиционного программирования. Усвоение этих простых понятий откроет перед вами целый мир разнообразных и сложных приложений. Тут все как у людей: в ДНК каждого человека на Земле есть всего четыре различных нуклеооснования, но все равно все мы разные, уникальные.

Если вы еще не знакомы с программированием, то в этой главе будет много новой для вас информации, и здесь вы напишете первые в своей жизни строки кода. Я не пытаюсь перегрузить ваш мозг фактами и цифрами, а лишь хочу максимально полно и ясно рассказать об основных элементах программирования, опираясь на примеры из повседневной жизни.

В этой главе я приведу высокоуровневый разбор элементов, из которых состоит программа. Познакомившись с принципами работы до того, как переходить непосредственно к написанию кода, вы как начинающий программист будете крепче стоять на ногах, лучше поймете материал, а хорошо запоминающиеся примеры лишь укрепят это понимание. Среди прочего в данной главе мы обсудим следующие вопросы:

- понятие переменных и способы их использования;
- назначение методов;
- классы и как они «превращаются» в объекты;
- как превратить сценарии C# в компоненты Unity;
- связь между компонентами и точечную нотацию.

Определение переменных

Начнем с простого вопроса: что такое переменная? В зависимости от точки зрения ответить на этот вопрос можно несколькими способами.

- **Концептуально** переменная — это мельчайшая базовая единица программирования, как атом для физического мира (умолчу о теории струн). Все начинается с переменных, и без них программы попросту не могут существовать.
- **Технически** переменная — небольшой фрагмент памяти вашего компьютера, который содержит присвоенное переменной значение. Каждая переменная знает, где хранится ее информация (адрес в памяти), свое значение и тип (это могут быть, например, числа, слова или списки).
- **По сути** переменная — контейнер. При необходимости вы можете самостоятельно создавать их, заполнять чем-то, перемещать, изменять содержимое и ссылаться на них. Иногда полезны даже пустые переменные.

В качестве аналогии переменной можно привести почтовые ящики из реальной жизни, наподобие таких (рис. 2.1).

Рис. 2.1

В них могут храниться письма, счета, фотографии вашей тети — все что угодно. Важно то, что содержимое почтового ящика может быть разным: они могут иметь имена, хранить информацию (собственно почту), а еще их содержимое можно менять, если у вас есть ключ.

Имена важны

Еще раз посмотрим на рис. 2.1. Если я попрошу вас пойти и открыть почтовый ящик, вы, скорее всего, захотите уточнить: «Какой именно?» Если бы я сказал более точно, например: «ящик Ивановых», «коричневый почтовый ящик» или «круглый почтовый ящик», то у вас было бы достаточно информации (или контекста) для выполнения моего поручения. Аналогично, создавая переменную, вы должны дать ей уникальное имя, по которому можно было бы обратиться к ней позже. Об особенностях правильного форматирования и описательного именования мы подробнее поговорим в главе 3.

Переменные — это «коробки»

Объявляя переменную и давая ей имя, вы словно создаете пустую коробку для значения, которое хотите сохранить. Для примера возьмем следующее простое математическое равенство:

```
2 + 9 = 11
```

Здесь все просто, но что, если бы мы хотели, чтобы число 9 было взято из переменной? Рассмотрим следующий блок кода:

```
myVariable = 9
```

Теперь вместо числа 9 мы можем использовать имя переменной `myVariable` везде, где требуется:

```
2 + myVariable = 11
```

Если вам интересно, есть ли у переменных другие правила или ограничения, то да. О них поговорим в следующей главе, поэтому пока не торопитесь.

Приведенный выше пример не является реальным кодом C#, но зато иллюстрирует возможности переменных и их применение в качестве ссылок на «коробку со значением». Далее мы начнем создавать собственные переменные, так что вперед!

Время действовать. Создаем переменную

Пожалуй, хватит теории. Создадим настоящую переменную в нашем сценарии `LearningCurve`.

1. Дважды щелкните на сценарии `LearningCurve`, чтобы открыть его в Visual Studio, затем добавьте в него строки 7, 12 и 14 (пока не думайте о синтаксисе, а просто перепишите указанные строки с этого снимка экрана (рис. 2.2)).

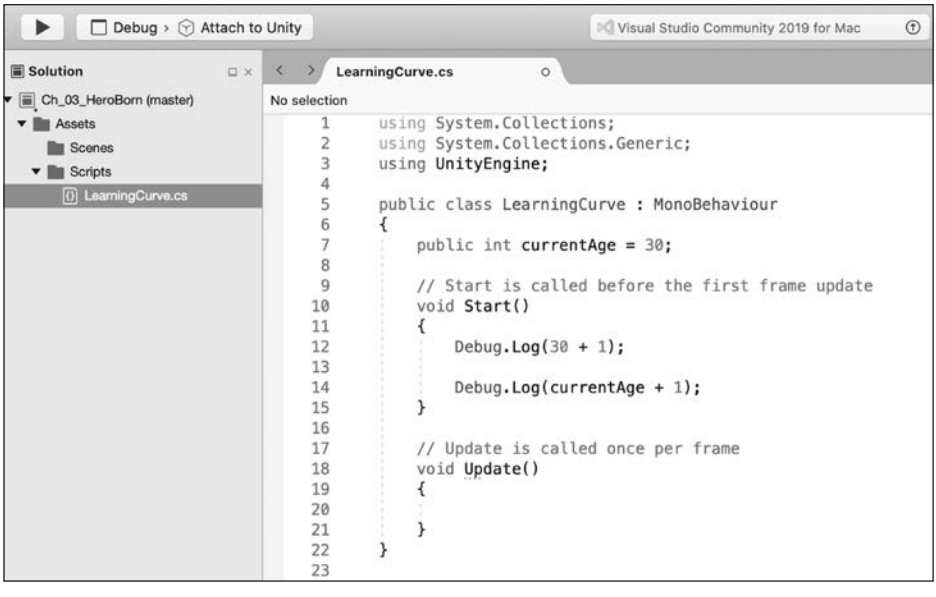


Рис. 2.2

2. Сохраните файл, используя сочетание клавиш ⌘+S на клавиатуре Mac или Ctrl+S на клавиатуре Windows.

Чтобы скрипты запускались в Unity, их нужно прикрепить к объекту `GameObject` на сцене. По умолчанию в проекте `HeroBorn` появились камера и направленный свет, который создает освещение сцены. Упростим задачу, прикрепив `LearningCurve` к камере.

1. Перетащите сценарий `LearningCurve.cs` на объект `Main Camera`.
2. Выберите объект `Main Camera`, чтобы он появился на панели Inspector, и убедитесь, что компонент `LearningCurve.cs` (Script) прикреплен правильно.

3. Нажмите кнопку Play и посмотрите, что появится на панели Console (рис. 2.3).

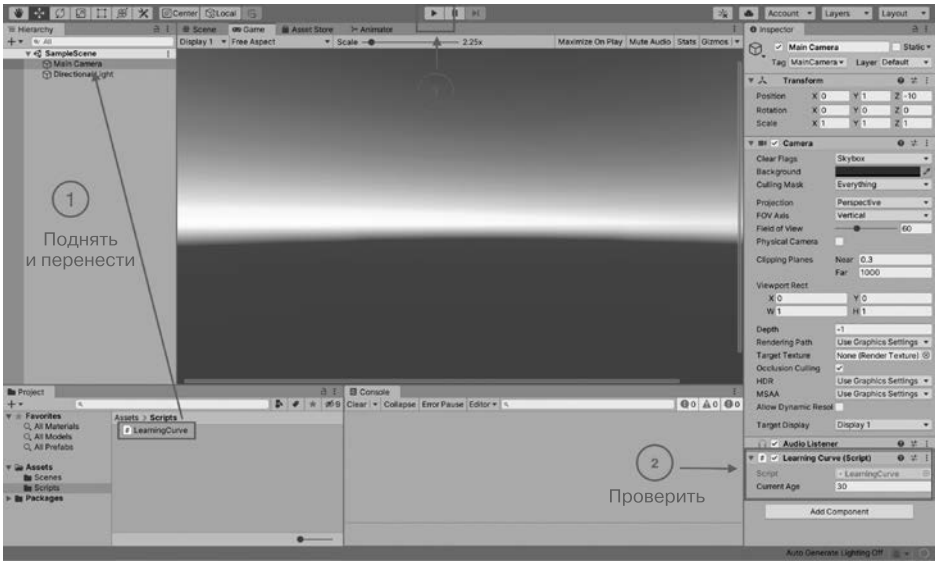


Рис. 2.3

Операторы `Debug.Log()` вывели в консоли результат вычисления простых математических выражений, которые мы заключили в круглые скобки. Как видно на снимке панели Console (рис. 2.4), выражение, в котором использовалась наша переменная, сработало так же, как и выражение с числами.

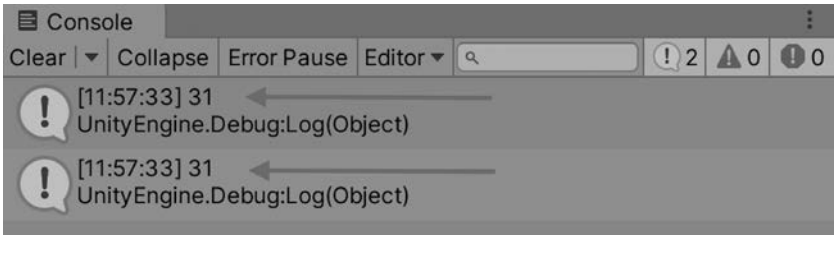


Рис. 2.4

В конце этой главы мы рассмотрим, как Unity преобразует сценарии C# в компоненты, но сначала попробуем изменить значение одной из наших переменных.

Время действовать. Меняем значение переменной

Поскольку в строке 7 мы объявили переменную `currentAge`, значение, которое в ней хранится, можно изменить. После этого уже новое значение будет подставляться туда, где переменная используется в коде. Посмотрим, как это работает.

1. Остановите игру, нажав кнопку Play, если сцена все еще запущена.
2. На панели Inspector измените значение параметра Current Age на `18`, снова запустите сцену и посмотрите на результат на панели Console (рис. 2.5).

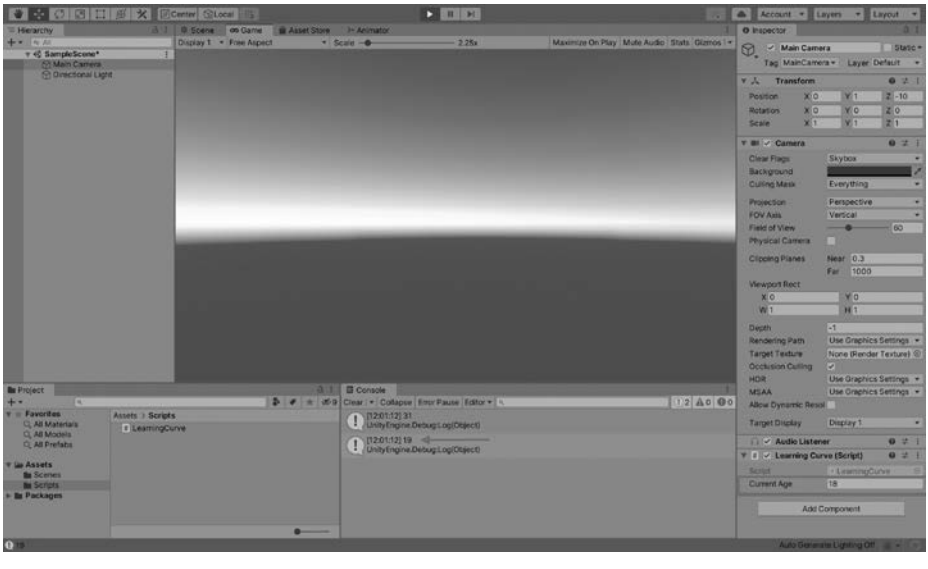


Рис. 2.5

Первый оператор по-прежнему вывел `31`, а второй — уже `19`, поскольку мы изменили значение нашей переменной.

Цель здесь заключалась не в прохождении синтаксиса переменных, а в демонстрации того, что переменные работают как контейнеры, которые мы один раз создаем, а затем ссылаемся на них в другом месте. Подробнее об этом поговорим в главе 3.

Теперь, зная, как создавать переменные в C# и присваивать им значения, мы готовы рассмотреть следующий важный элемент программирования: методы!

Понятие метода

Переменные могут лишь хранить присвоенные им значения. Эта функция невероятно важна, но сама по себе не слишком полезна с точки зрения создания серьезных приложений. Итак, как нам создать в коде какие-либо действия и управлять поведением? Ответ прост: с помощью методов.

Прежде чем я расскажу, что такое методы и как их применять, нужно немного прояснить один момент из терминологии. В мире программирования термины «метод» и «функция» часто используются как синонимы, особенно в Unity. Поскольку C# — объектно-ориентированный язык (в главе 5 мы рассмотрим, что означает это понятие), на протяжении всей книги я буду употреблять термин «метод», чтобы соответствовать стандартным рекомендациям C#.

Когда в Scripting Reference или в любой другой документации вы встретите слово «функция», это будет то же самое, что я здесь называю методом.

Методы — это движущая сила

Как и в случае с переменными, определение методов в программировании может быть или утомительно длинным, или опасно кратким. Снова подойдем к термину с трех сторон.

- **Концептуально** методы — это то, как в приложении выполняется работа.
- **Технически** метод — блок кода, содержащий операторы, которые срабатывают, когда метод вызывается по имени. Методы могут

принимать аргументы (также называемые параметрами), которые можно использовать внутри области действия метода.

- **По сути** метод — контейнер для набора инструкций, выполняющихся при каждом его вызове. Эти контейнеры также могут принимать в качестве входных данных переменные, на которые можно ссылаться только внутри самого метода.

Вместе взятые, методы образуют костяк любой программы, который связывает остальные элементы между собой и формирует общую структуру.

Методы — это тоже «коробки»

Возьмем тривиальный пример сложения двух чисел, чтобы понять концепцию. При написании сценария вы, по сути, пишете строки кода, которые компьютер должен выполнить по порядку. Когда вам впервые нужно сплюсовать два числа, вы можете просто взять и сложить их, как показано ниже:

```
someNumber + anotherNumber
```

Но затем оказывается, что эти числа нужно сложить где-то еще. Вместо того чтобы копировать и вставлять один и тот же код, что приводит к появлению неаккуратного кода (или так называемого спагетти-кода) и чего следует избегать любой ценой, вы можете создать именованный метод, который будет выполнять сложение:

```
AddNumbers
{
    someNumber + anotherNumber
}
```

Теперь метод `AddNumbers` хранится в памяти как переменная, но вместо значений содержит набор инструкций. Использование имени метода (или его вызов) в любом месте сценария позволяет выполнить сохраненные в нем инструкции, не прибегая к необходимости повторно писать что-либо.

Если вы в какой-то момент обнаружите, что пишете одни и те же строки кода снова и снова, то наверняка этот код можно упростить или объединить повторяющиеся действия в методы.

Повторное написание одного и того же кода приводит к явлению, которое программисты в шутку называют спагетти-кодом, поскольку он получается беспорядочным. В связи с этим часто можно услышать о принципе Don't Repeat Yourself (DRY), то есть «Не повторяйся», и о нем следует помнить.

Как и раньше, рассмотрев новую концепцию в псевдокоде, лучше всего реализовать ее самостоятельно и довести до ума, что мы и сделаем ниже.

Время действовать. Создаем простой метод

Снова откроем файл `LearningCurve` и посмотрим на работу методов в C#. Как и в примере с переменными, перепишите код в свой сценарий, чтобы получить то же, что и на рис. 2.6. Я удалил предыдущий пример кода, чтобы не было беспорядка, но вы можете и сохранить его в своем сценарии для справки.

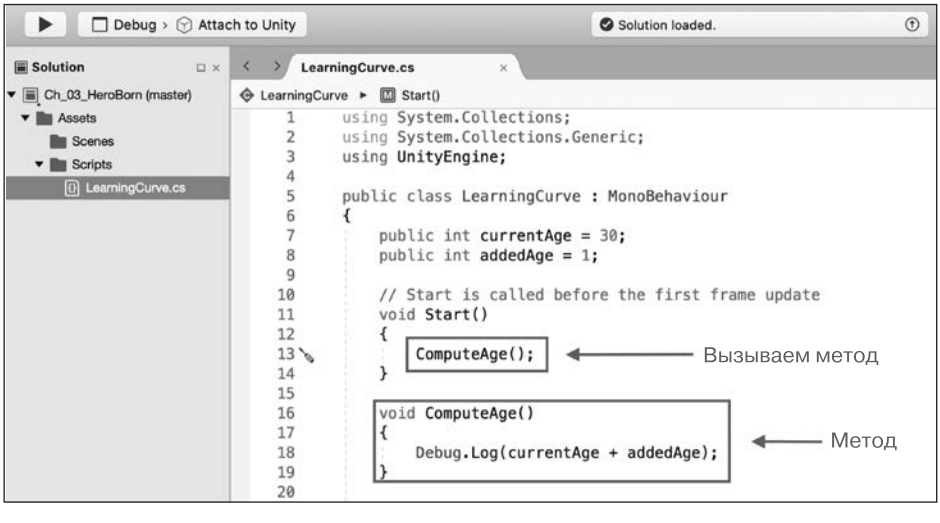


Рис. 2.6

1. Откройте сценарий `LearningCurve` в Visual Studio и добавьте строки 8, 13 и 16–19.
2. Сохраните файл, а затем вернитесь в Unity и нажмите кнопку Play, чтобы посмотреть, что появилось в консоли.

Мы определили первый метод в строках с 16-й по 19-ю и вызвали его в строке 13. Теперь везде, где мы будем вызывать метод `AddNumbers()`, программа станет суммировать две переменные и выводить их в консоль, даже если их значения изменятся (рис. 2.7).

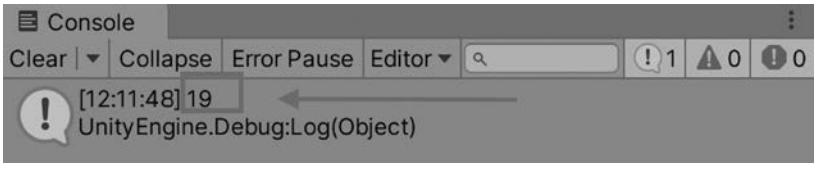


Рис. 2.7

Самостоятельно попробуйте задавать разные значения переменных на панели Inspector, чтобы проверить, как это работает. Подробнее о самом синтаксисе кода, который вы только что написали, поговорим в следующей главе.

Теперь, получив базовое понимание работы методов, мы готовы перейти к самой серьезной теме в области программирования — классам!

Знакомство с классами

Мы уже знаем, что переменные хранят информацию и методы выполняют действия, но пока наш инструментарий программирования все еще несколько ограничен. Нам нужна возможность создать суперконтейнер со своими переменными и методами, на которые можно ссылаться из самого контейнера. Это и будут классы.

- **Концептуально** класс хранит некую информацию, действия и поведение внутри одного контейнера. Эти элементы могут даже общаться друг с другом.

- **Технически** классы — структуры данных. Они могут содержать переменные, методы и другую программную информацию, и на все эти элементы можно ссылаться при создании объекта класса.
- **По сути** класс — некий шаблон. Он устанавливает правила для любого объекта (называемого экземпляром), созданного с помощью этого шаблона.

Вы наверняка поняли, что классы есть не только в Unity, но и в реальном мире. Далее мы рассмотрим наиболее распространенный класс Unity и то, как классы работают в реальных условиях.

Стандартный класс Unity

Прежде чем задаться вопросом, как выглядит класс в C#, отметим, что всю эту главу вы уже работали с классом. По умолчанию каждый сценарий, созданный в Unity, является классом, и на это намекает ключевое слово `class` в строке 5:

```
public class LearningCurve: MonoBehavior
```

Имя `MonoBehavior` просто означает, что этот класс может быть присоединен к `GameObject` в сцене Unity. В C# классы могут существовать сами по себе, что мы увидим, когда будем писать автономные классы в главе 5.

Термины «сценарий» и «класс» иногда используются в ресурсах Unity как синонимы. Для единообразия я буду называть файлы C# сценариями, если они прикреплены к объектам GameObject, и классами, если они автономны.

Классы — это шаблоны

В последнем примере рассмотрим некое почтовое отделение. Это отдельная автономная среда, у которой есть свойства, например физический адрес (переменная) и возможность выполнять действия, такие как отправка вашего письма (методы).

Класс `PostOffice` — это отличный пример потенциального класса, который мы можем описать в следующем псевдокоде:

```
PostOffice
{
    // Переменные
    Address = "1234 Letter Opener Dr."

    // Методы
    DeliverMail()
    SendMail()
}
```

Основная польза здесь заключается в том, что, когда информация и поведение следуют заранее определенному шаблону, становится возможным организовать сложные действия и межклассовое общение.

Например, если у нас есть другой класс, который хочет отправить письмо через наш класс, то ему не придется гадать, куда идти, чтобы это сделать. Можно просто вызвать функцию `SendMail` из класса `PostOffice` следующим образом:

```
PostOffice.SendMail()
```

Кроме того, мы можем получить адрес почтового отделения, чтобы знать, куда отправлять письма:

```
PostOffice.Address
```

Если вас интересует, откуда взялись точки между словами (точечная нотация), то мы рассмотрим это в конце главы — держитесь крепче.

Теперь базовый инструментарий программирования в нашем распоряжении (ну, по крайней мере, его теоретическая часть). Далее мы углубимся в синтаксис и практическое использование переменных, методов и классов.

Работа с комментариями

Вы могли заметить, что в сценарии `LearningCurve` есть две строки серого текста (строка 10 на рис. 2.6), начинающиеся с двух косых черт, которые

появились в сценарии сразу в момент его создания. Это комментарии к коду — очень эффективный и в то же время простой инструмент программиста.

В C# есть несколько способов создания комментариев, а в Visual Studio (и других редакторах кода) их создание часто упрощается с помощью горячих клавиш.

Некоторые специалисты не назвали бы комментарии важным элементом программирования, но я вынужден с этим не согласиться. Правильное комментирование кода значимой информацией — одна из самых важных привычек, которой должен обзавестись начинающий программист.

Практичные косые

В сценарии `LearningCurve` есть однострочные комментарии:

```
// это однострочный комментарий
```

Visual Studio не видит в коде строк, начинающихся с двух косых черт (без пробелов между ними), поэтому вы можете использовать их сколько угодно раз.

Многострочные комментарии

Из названия понятно, что однострочные комментарии охватывают лишь одну строку кода. Если вам нужны многострочные комментарии, то нужно использовать косую черту и звездочку в качестве открывающих и закрывающих символов вокруг текста комментария:

```
/* это
    многострочный комментарий * /
```

Вы также можете комментировать и раскомментировать блоки кода, выделяя их и используя сочетание клавиш ⌘+? в macOS и Ctrl+K+C в Windows.

Видеть примеры комментариев — это хорошо, а использовать их в своем коде еще лучше. Начать комментировать никогда не рано!

Время действовать. Добавляем комментарии

В Visual Studio также есть удобная функция автоматического создания комментариев. Введите три косые черты в строке, предшествующей любой строке кода (переменные, методы, классы и т. д.), и в результате появится блок комментариев `summary`. Откройте `LearningCurve` и добавьте три косые черты над методом `ComputeAge()` (рис. 2.8).

```
16      /// <summary>
17      /// Computes a modified age integer
18      /// </summary>
19      void ComputeAge()
20      {
21          Debug.Log(currentAge + addedAge);
22      }
```

Рис. 2.8

Появится трехстрочный комментарий с описанием метода, созданный Visual Studio для данного имени метода. Само описание находится между двумя тегами `<summary>`. Разумеется, вы можете изменить текст или добавить новые строки, нажав клавишу Enter, как и в текстовом документе, но теги при этом удалять нельзя.

Польза подробных комментариев очевидна, если вы хотите описать созданный вами метод. Если вы использовали комментарий с тройной косой чертой, то в дальнейшем можете навести указатель мыши на имя метода в любом месте кода, и Visual Studio покажет ваш комментарий (рис. 2.9).

```
11      // Start is called before the first frame update
12      void Start()
13      {
14          ComputeAge();
15      }
16
17      // U                                    frame
18      void Update()
19      {
```

void LearningCurve.ComputeAge()
Computes a modified age integer

Рис. 2.9

Осталось понять, как все, что мы узнали в этой главе, применяется в игровом движке Unity. И именно об этом поговорим в следующем разделе!

Собираем все вместе

Когда мы разобрались со всеми нужными элементами по отдельности, пришло время немного поработать с Unity, прежде чем завершать эту главу. В частности, нам нужно узнать о том, как Unity обрабатывает сценарии C#, прикрепленные к объектам `GameObject`. В этом примере мы продолжим использовать наш сценарий `LearningCurve` и объект `Main Camera`.

Сценарии превращаются... в элегантные компоненты

Все компоненты `GameObject` — это сценарии, написанные вами или добрыми людьми из команды Unity. Однако нам не нужно редактировать специфичные для Unity компоненты, такие как `Transform`, и их сценарии.

Когда созданный вами сценарий помещается в объект `GameObject`, он становится еще одним компонентом этого объекта и появляется на панели Inspector. С точки зрения Unity он работает так же, как и любой другой компонент, а его публичные переменные, показанные внутри компонента, можно изменять в любое время. Несмотря на невозможность редактировать встроенные компоненты Unity, мы все равно можем обращаться к их свойствам и методам, что делает их эффективными инструментами разработки.

Unity также автоматически настраивает удобочитаемость, когда сценарий становится компонентом. Вы могли заметить, что, когда мы добавили сценарий LearningCurve к объекту Main Camera, Unity отобразил его имя как Learning Curve, а переменная currentAge превратилась в параметр Current Age.

В предыдущем пункте «Время действовать» мы уже пробовали менять значение переменной на панели Inspector, и теперь пора более подробно остановиться на том, как это работает. Есть две ситуации, в которых вы можете изменить значение свойства:

- в режиме воспроизведения игры;
- в режиме разработки.

Изменения, внесенные в режиме воспроизведения, вступают в силу немедленно в реальном времени, что отлично подходит для тестирования и настройки игрового процесса. При этом любые изменения, внесенные в режиме игры, будут утрачены, когда вы остановите игру и вернетесь в режим разработки.

Когда вы находитесь в режиме разработки, любые изменения, которые вы вносите в переменные, будут сохранены в Unity. Это значит, что если вы выйдете из Unity, а затем перезапустите его, то изменения сохранятся.

Изменения, которые вы вносите в значения на панели Inspector, не влияют на сценарий, но в режиме воспроизведения заменяют собой любые значения, которые вы назначили в сценарии.

Если вам нужно отменить какие-либо изменения, внесенные на панели Inspector, вы можете сбросить сценарий до значений по умолчанию (иногда называемых начальными). Щелкните на значке с тремя вертикальными точками справа от любого компонента, а затем выберите команду Reset, как показано на рис. 2.10.

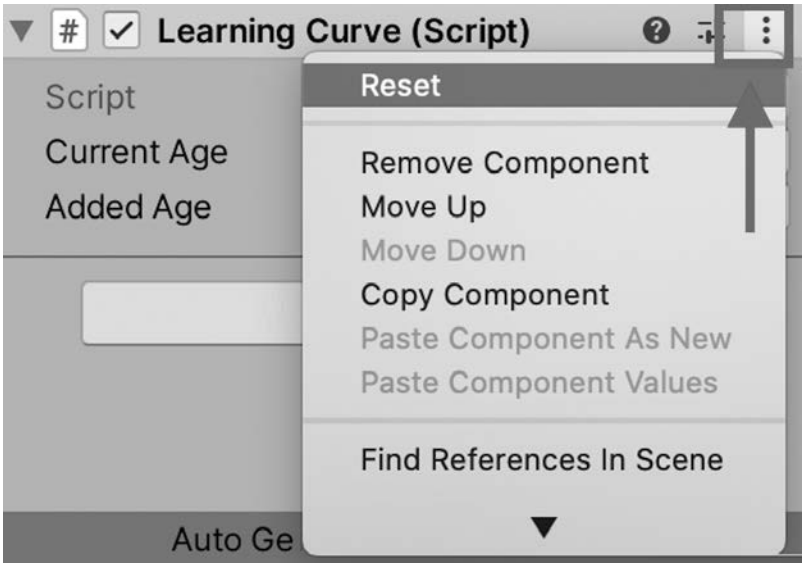


Рис. 2.10

Это должно вас успокоить, поскольку, если вы вдруг заигрались со значениями переменных, всегда есть возможность сброса.

Рука помощи от MonoBehavior

Если сценарии C# являются классами, то откуда Unity знает, нужно делать сценарий компонентом или нет? Ответ прост: `LearningCurve` (и любой сценарий, созданный в Unity) наследуется от `MonoBehavior` (другого класса). Для Unity это означает, что класс C# можно преобразовать в компонент.

Тема наследования классов на данном этапе будет для вас сложновата. Выразимся так: есть класс `MonoBehaviour`, который предоставляет несколько своих переменных и методов классу `LearningCurve`. В главе 5 мы рассмотрим наследование классов подробно.

Используемые нами методы `Start()` и `Update()` принадлежат классу `MonoBehavior`, который Unity запускает автоматически для любого сценария, прикрепленного к `GameObject`. Метод `Start()` запускается один раз при запуске сцены, а метод `Update()` выполняется в каждом кадре (в зависимости от частоты кадров вашего компьютера).

Теперь, когда вы неплохо познакомились с документацией Unity, предлагаю вам решить небольшую дополнительную задачу, которую я для вас подготовил!

Испытание героя. Ищем MonoBehavior в Scripting API

Теперь вам пора освоиться с документацией Unity самостоятельно, и что может быть лучше, чем поискать некоторые из распространенных методов `MonoBehavior`:

- попробуйте поискать методы `Start()` и `Update()` в `Scripting API`, чтобы лучше понять, что и когда они делают в Unity;
- если не боитесь, то почитайте о классе `MonoBehavior` еще и в руководстве, где о нем рассказано более подробно.

Прежде чем слишком глубоко погрузиться в наше приключение в мире программирования на C#, нам нужно рассмотреть последний важный элемент — связь между классами.

Связь между классами

До сих пор я описывал классы и, как следствие, компоненты Unity как отдельные автономные сущности. Однако в действительности они глубоко переплетены. Сложно создать серьезное приложение, в котором не было бы никакого взаимодействия или связи между классами.

Если вы еще не забыли предыдущий пример с почтовым отделением, то в нем использовали точки, чтобы сослаться на классы, переменные и методы. Если представить классы как каталоги информации, то точечная нотация — это инструмент индексации:

```
PostOffice.Address
```

С помощью точечной нотации можно получить доступ к любым переменным, методам или другим типам данных в классе. Это применимо и к вложенной информации или информации о подклассах, но этими вопросами мы займемся позже, в главе 5.

Точечная нотация также определяет связь между классами. Когда классу требуется информация о другом классе или он хочет выполнить один из его методов, используется точечная нотация:

```
PostOffice.DeliverMail()
```

Точечная нотация иногда упоминается как «оператор (.)», поэтому не расстраивайтесь, если увидите в документации нечто подобное.

Если точечная нотация вам пока не нравится, то не волнуйтесь, еще притретесь. Это кровоток программы, передающий информацию и контекст туда, где это необходимо.

Подведем итоги

Мы много чего рассмотрели на этих нескольких страницах, но понимание фундаментальных понятий, таких как переменные, методы и классы, заложит прочную основу для дальнейшего развития. Помните, что у всех этих элементов есть настоящие аналоги в реальном мире. Переменные

хранят значения, как почтовые ящики хранят письма. Методы, словно рецепты, хранят инструкции, которым нужно следовать для достижения заранее определенного результата. Классы — это шаблоны, подобные чертежам. Вы не можете построить дом без хорошо продуманного дизайна, которому обязательно следовать, чтобы ничего не упало.

В оставшейся части книги мы рассмотрим синтаксис C# с нуля и в следующей главе начнем с подробностей о том, как создавать переменные, управлять типами значений и работать с простыми и сложными методами.

Контрольные вопросы. Основные элементы C#

1. Каково основное назначение переменных?
2. Какую роль в сценариях играют методы?
3. Как сценарий превратить в компонент?
4. Для чего нужна точечная нотация?

3 Погружение в переменные, типы и методы

Поначалу в освоении любого языка программирования возникает фундаментальная проблема — вы понимаете набираемые слова, но не их назначение. Обычно в таких ситуациях возникает парадокс, но программирование — особый случай.

C# написан на английском. Несоответствие между словами, которые вы используете в своей жизни, и кодом в Visual Studio возникает из-за отсутствия контекста, который всегда разный. Вы знаете, как произносить и записывать слова, используемые в C#, но не знаете, где, когда, почему и, что наиболее важно, как именно они превращаются в язык программирования.

В этой главе мы отойдем от теории программирования и сделаем первые шаги на нашем пути к реальному программированию. Мы поговорим о принятых правилах форматирования, методах отладки и составлении более сложных примеров переменных и методов. Нам нужно осветить много вопросов, но к моменту, когда вы дойдете до контрольных вопросов, мы рассмотрим такие высокоуровневые вопросы, как:

- пишем на C# правильно;
- отладка вашего кода;
- объявление переменных;
- использование модификаторов доступа;
- понятие области видимости переменных;
- работа с методами;
- анализ распространенных методов Unity.

Пишем на C# правильно

Строки кода работают как предложения, то есть в них должен быть некий разделительный или конечный символ. Каждая строка C#, называемая оператором, обязательно **должна** заканчиваться точкой с запятой, чтобы компилятор мог понять, где заканчивается одна команда и начинается другая.

Однако есть одна особенность, о которой вам нужно знать. В отличие от обычного письменного языка, с которым вы знакомы, оператор C# не обязательно должен располагаться в одной строке. Пробелы и символы переноса строки компилятором игнорируются. Например, простую переменную можно объявить так:

```
public int firstName = "Harrison";
```

а можно и так:

```
public
int
firstName
=
"Harrison";
```

Оба эти фрагмента кода одинаково приемлемы для Visual Studio, но второй вариант крайне не одобряется сообществом разработчиков программного обеспечения, поскольку такой код становится чрезвычайно трудным для чтения. Идея состоит в том, что писать программы нужно как можно эффективнее и понятнее.

Бывают моменты, когда оператор получается слишком длинным и не умещается в одной строке, но это редкость. Но если так произошло, то отформатируйте данный оператор таким образом, чтобы и другие люди могли его понять, и не забудьте точку с запятой.

Второе правило форматирования, которое нужно записать у себя на подкорке, — использование фигурных скобок. У всех методов, классов и интерфейсов после их объявления должна идти пара фигурных скобок. Мы поговорим о каждом из этих элементов более подробно позже,

но продумать стандартное форматирование нужно заблаговременно. Традиционная практика в C# — писать каждую скобку на новой строке, как показано ниже:

```
public void MethodName()
{

}
```

Однако, когда вы создаете в редакторе Unity новый сценарий или открываете документацию Unity, первая фигурная скобка оказывается в той же строке, что и объявление:

```
public void MethodName() {

}
```

Убиваться из-за этого не стоит, но важно соблюдать принятые принципы. В чистом коде C# всегда каждая скобка занимает новую строку, в то время как примеры C#, связанные с Unity и разработкой игр, чаще оформлены так, как показано во втором образце.

Хороший, последовательный стиль форматирования имеет огромное значение, когда вы только начинаете программировать, равно как и возможность наконец увидеть плоды своей работы. В следующем разделе мы поговорим о том, как выводить переменные и информацию прямо в консоли Unity.

Отладка кода

Когда мы будем работать над практическими примерами, нам понадобится способ распечатать информацию и отзывы на панели консоли в редакторе Unity. В программировании это называется отладкой, и в C# и в Unity есть вспомогательные методы, упрощающие данный процесс для разработчиков. Каждый раз, когда я попрошу вас отладить или вывести что-либо на экран, используйте один из методов, которые описаны ниже.

- Для простого текста или отдельных переменных задействуйте стандартный метод `Debug.Log()`. Текст должен быть заключен в кавычки,

а переменные можно использовать напрямую, без добавления символов. Например:

```
Debug.Log("Text goes here.");
Debug.Log(yourVariable);
```

- Для более сложной отладки используйте метод `Debug.LogFormat()`. Он позволяет размещать переменные внутри выводимого текста через заполнители. Они отмечены парой фигурных скобок, каждая из которых содержит индекс. Индекс — это обычное число, начинающееся с 0 и последовательно увеличивающееся на 1.

В следующем примере заполнитель `{0}` заменяется значением переменной `variable1`, `{1}` — значением `variable2` и т. д.:

```
Debug.LogFormat("Text goes here, add {0} and {1} as variable
placeholders", variable1, variable2);
```

Вы могли заметить, что в наших методах отладки мы используем точечную нотацию, и это не случайность! `Debug` — класс, а `Log()` и `LogFormat()` — методы из данного класса, которые мы можем задействовать. Подробнее об этом — в конце главы.

Теперь, зная о возможностях отладки, мы можем смело двигаться дальше и глубже погрузиться в то, как объявляются переменные, а также в различные способы использования синтаксиса.

Объявление переменных

В предыдущей главе мы посмотрели, как записываются переменные, и коснулись высокоуровневого функционала, который они предоставляют. Однако нам все еще не хватает синтаксиса, который позволил бы реализовать этот функционал. Переменные не просто появляются в верхней части сценария C#. Их нужно объявлять в соответствии с определенными правилами и требованиями. На самом базовом уровне оператор переменной должен удовлетворять следующим требованиям:

- необходимо указать тип данных, которые будет хранить переменная;
- переменная должна иметь уникальное имя;

- если мы присваиваем переменной значение, то оно должно соответствовать указанному типу;
- объявление переменной должно заканчиваться точкой с запятой.

Совокупность этих правил дает следующий синтаксис:

```
dataType uniqueName = value;
```

Переменным нужны уникальные имена, чтобы избежать конфликтов со словами, уже используемыми в самом C#, которые называются ключевыми. Вы можете найти полный список защищенных ключевых слов, пройдя по ссылке docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/.

Этот синтаксис прост, аккуратен и эффективен. Однако язык программирования не был бы полезен в долгосрочной перспективе, если бы существовал только один способ создать что-то используемое столь часто, как переменные. В сложных приложениях и играх встречаются разные варианты применения и сценарии, каждый из которых написан с использованием уникального синтаксиса C#.

Объявление типа и значения

Наиболее распространенный сценарий создания переменных — тот, в котором вся необходимая информация указывается сразу при объявлении. Например, если бы мы знали возраст игрока, то написали бы нечто такое:

```
int currentAge = 32;
```

В этой строке выполнены все основные требования:

- указан тип данных — `int` (сокращение от `integer`);
- используется уникальное имя `currentAge`;
- `32` — целое число, соответствующее указанному типу данных;
- оператор заканчивается точкой с запятой.

Однако бывают случаи, в которых нужно объявить переменную, не зная ее значения заранее. Об этом мы поговорим в следующем подразделе.

Объявление типа без значения

Рассмотрим другой случай: вам известны тип данных, которые вы хотите хранить в переменной, и ее имя, а значение неизвестно. Оно будет вычислено и присвоено где-нибудь еще, но вам все равно нужно объявить переменную в начале сценария.

Эта ситуация идеально подходит для вот такого объявления:

```
int currentAge;
```

Здесь определены только тип (`int`) и уникальное имя (`currentAge`), и такой оператор тоже работает и соответствует правилам. Без присвоенного значения переменная получит значение по умолчанию, соответствующее типу переменной. В данном случае `currentAge` станет равно `0`, что соответствует типу `int`. Когда фактическое значение будет известно, его можно легко установить в отдельном операторе, указав имя переменной и присвоив ей значение:

```
currentAge = 32;
```

Полный список всех типов C# и их значений по умолчанию можно найти по адресу docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/default-values.

Здесь вы можете спросить, почему у этих переменных не было ключевого слова `public`, называемого *модификатором доступа*, которое мы видели в предыдущих примерах сценариев. Дело в том, что на тот момент у нас не было почвы для разговора о данном модификаторе. Теперь пришло время вернуться к этому моменту.

Использование модификаторов доступа

Итак, базовый синтаксис нам понятен, поэтому перейдем к более тонким подробностям операторов переменных. Поскольку мы читаем код слева направо, имеет смысл начать подробное изучение переменных с ключевого слова, которое традиционно идет первым, — с модификатора доступа.

Взгляните на переменные, которые мы использовали в предыдущей главе в сценарии `LearningCurve`, и вы увидите, что перед их объявлениями добавлено дополнительное ключевое слово `public`. Это модификатор доступа к переменной, то есть некий параметр безопасности, определяющий, кто и что может получить доступ к информации о переменной.

Любая переменная, не имеющая модификатора public, по умолчанию получает модификатор private и не будет отображаться на панели Inspector в Unity.

Если мы добавим модификатор, то рецепт синтаксиса, который мы собрали в начале этой главы, будет выглядеть следующим образом:

```
accessModifier dataType uniqueName = value;
```

Хотя явные модификаторы доступа при объявлении переменной не всегда нужны, начинающим программистам было бы хорошо завести привычку их добавлять. Это лишнее слово значительно повышает читаемость и профессионализм вашего кода.

Выбор уровня безопасности

В C# есть четыре основных модификатора доступа, но вам, как новичкам, чаще всего предстоит работать с двумя из них:

- `public` — переменная доступна для любого сценария без ограничений;
- `private` — переменная доступна только в классе, в котором создана (он называется содержащим классом (containing class)).

Любая переменная без модификатора доступа по умолчанию является частной.

Есть еще два более сложных модификатора:

- `protected` — доступна из содержащего класса или производных от него типов;
- `internal` — доступна только в текущей сборке.

Каждый из этих модификаторов предполагает свои варианты использования, но пока мы не дошли до более сложных глав, о модификаторах `protected` и `internal` можно временно забыть.

Существует еще два комбинированных модификатора, но в этой книге мы не будем их использовать. Подробнее о них можно почитать, пройдя по ссылке docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/access-modifiers.

Теперь опробуем эти модификаторы доступа!

Время действовать. Делаем переменную частной

Как и в реальной жизни, одни данные необходимо защищать, а другие можно передавать конкретным людям. Если в вашем проекте нет необходимости изменять переменную на панели Inspector или обращаться к ней из других сценариев, то такой переменной можно задать модификатор `private`.

Подредактируем сценарий `LearningCurve`, выполнив следующие действия.

1. Измените модификатор доступа переменной `currentAge` с `public` на `private` и сохраните файл.
2. Вернитесь в Unity, выберите объект `Main Camera` и посмотрите, что изменилось в разделе `LearningCurve`.

Поскольку переменная `currentAge` теперь является частной, она больше не отображается на панели Inspector и доступна только в сценарии `LearningCurve` (рис. 3.1). Если мы нажмем кнопку Play, то сценарий будет работать точно так же, как и раньше.

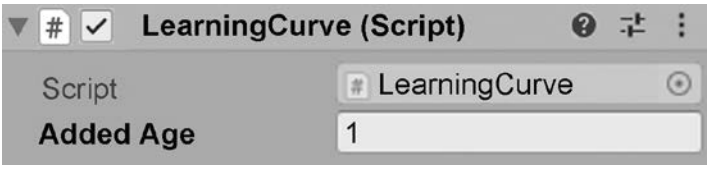


Рис. 3.1

Мы уже много чего знаем о переменных, но нам все еще нужно узнать больше о том, какие типы данных они могут хранить. И в следующем разделе мы поговорим именно о типах данных.

Работа с типами

Присвоение переменной определенного типа — важный выбор, влияющий в дальнейшем на каждое ее взаимодействие на протяжении всего ее жизненного цикла. Поскольку C# — так называемый строго типизированный или типобезопасный язык, абсолютно каждая переменная должна иметь определенный тип данных. Это значит, что существуют четкие правила при выполнении операций с определенными типами, а также правила преобразования одного типа переменной в другой.

Простые встроенные типы

Все типы данных в C# наследуются (или являются *производными*, если говорить как программисты) от общего предка: `System.Object`. Эта иерархия, называемая **системой общих типов** (Common Type System, CTS), означает, что разные типы имеют много общих функций. В табл. 3.1 представлены некоторые из наиболее распространенных вариантов типов данных и хранимые ими значения.

Таблица 3.1

Тип	Содержимое переменной
`Int`	Целые числа, например 42
`Float`	Число с плавающей запятой, например 3,14
`String`	Набор символов в кавычках, например «абыр валг»
`Bool`	Логическое значение — true или false

Помимо указания на значения, которые может хранить переменная, типы содержат дополнительную информацию о себе:

- необходимое для хранения пространство;
- минимальные и максимальные значения;

- допустимые операции;
- расположение в памяти;
- доступные методы;
- базовый (производный) тип.

Если этот объем информации кажется вам слишком большим, то сделайте глубокий вдох. Работа со всеми типами, которые есть в C#, — прекрасный пример использования документации вместо запоминания. Довольно скоро применение даже самых сложных пользовательских типов станет для вас простым и родным.

Вы можете найти полный список всех встроенных типов C# и их спецификации по адресу docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/types/.

Чтобы не застрять с этим огромным списком типов, лучше поэкспериментировать с ними. В конце концов, лучший способ узнать нечто новое — это сделать что-то, сломать, а затем научиться исправлять.

Время действовать. Экспериментируем с типами

Откройте сценарий `LearningCurve` и добавьте в него новую переменную каждого типа, которые мы перечислили выше. Имена и значения можете выбрать сами, но не забудьте добавить им модификатор `public`, чтобы они появились на панели Inspector. Если вам нужно вдохновение, то взгляните на мой код, который показан на рис. 3.2.

При работе со строковыми типами присваиваемое текстовое значение должно находиться внутри пары двойных кавычек, а значения с плавающей запятой должны заканчиваться строчной буквой f — в примере это видно.

Теперь мы видим все использованные типы переменных. Обратите внимание на переменную типа `bool`, которая отображается в Unity в виде флажка (если галочка есть — переменная получает значение `true`, если нет, то `false`) (рис. 3.3).

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class LearningCurve : MonoBehaviour
{
    // Integer variables
    private int currentAge = 30;
    public int addedAge = 1;

    public float pi = 3.14f;
    public string firstName = "Harrison";
    public bool isAuthor = true;

    // Start is called before the first frame update
    void Start()
    {
        ComputeAge();
    }

    // Update is called once per frame
    void Update()
    {

    }

    /// <summary>
    /// Computes a modified age integer
    /// </summary>
    void ComputeAge()
    {
        Debug.Log(currentAge + addedAge);
    }
}

```

Рис. 3.2

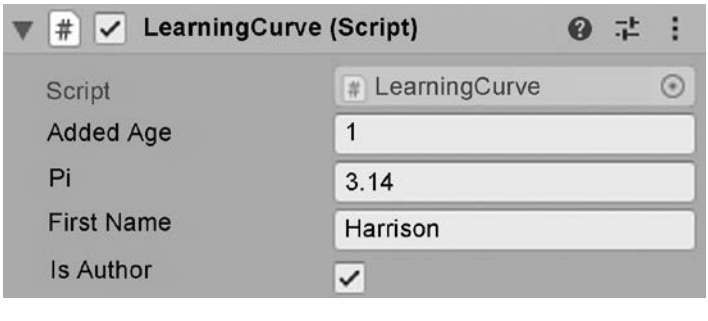


Рис. 3.3

Прежде чем мы перейдем к преобразованиям, рассмотрим распространенный и эффективный способ применения строкового типа данных — а именно, создание строк, в которые включаются переменные.

Время действовать. Создаем интерполированные строки

Если числовые типы ведут себя так, как нас учили на уроках математики в начальной школе, то со строками дело обстоит иначе. Мы можем вставлять переменные и литералы непосредственно в текст, если начнем его с символа `$`, что называется строковой интерполяцией. Интерполированные значения добавляются в фигурные скобки, как при использовании метода `LogFormat()`. Создадим простую интерполированную строку в сценарии `LearningCurve`, чтобы попробовать это в действии.

Выведите интерполированную строку внутри метода `Start()` сразу после вызова метода `ComputeAge()` (рис. 3.4).

```
// Start is called before the first frame update
void Start()
{
    ComputeAge();

    Debug.Log($"A string can have variables like {firstName} inserted directly!");
}
```

Рис. 3.4

Благодаря фигурным скобкам значение переменной `firstName` выводится внутри интерполированной строки (рис. 3.5).

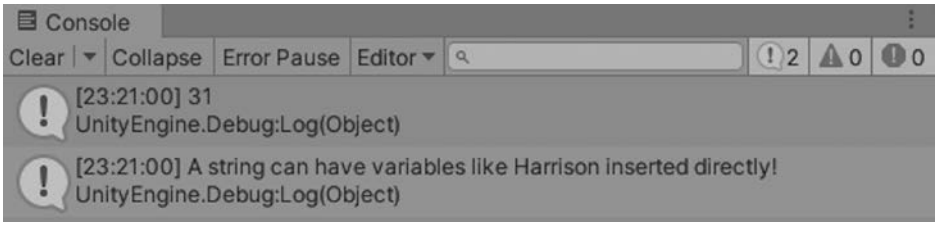


Рис. 3.5

Мы также можем создавать интерполированные строки с помощью оператора `+`, к которому перейдем сразу после того, как поговорим о преобразованиях типов.

Преобразование типов

Мы уже видели, что переменные могут содержать только значения своих объявленных типов, но иногда приходится комбинировать переменные разных типов. В терминологии программирования они называются преобразованиями и бывают двух основных видов.

- Неявные преобразования происходят автоматически, обычно когда меньшее значение помещается в другой тип переменной без округления. Например, любое целое число можно неявно преобразовать в `double` или `float`, не прилагая дополнительных усилий:

  ```
  float implicitConversion = 3;
  ```

- Явные преобразования необходимы в случаях, когда во время преобразования существует риск потерять некую информацию. Например, если бы мы хотели преобразовать переменную типа `double` в `int`, то нам пришлось бы явно преобразовать ее, добавив целевой тип в круглых скобках перед значением, которое мы хотим преобразовать. Так мы сообщаем компилятору, что знаем о предстоящей потере данных (или точности).

В этом явном преобразовании число `3.14` будет округлено до `3`, то есть дробная часть отбрасывается:

```
int explicitConversion = (int)3.14;
```

В C# есть встроенные методы для явного преобразования значений в простые типы. Например, любой тип можно преобразовать в строковое значение с помощью метода ToString(), а метод Convert может обрабатывать более сложные преобразования. Подробнее об этом можно почитать в разделе Methods, пройдя по ссылке docs.microsoft.com/ru-ru/dotnet/api/system.convert?view=netframework-4.7.2.

Итак, мы узнали, что у типов есть правила взаимодействия, операций и преобразования. Но что делать, когда нам нужно сохранить переменную неизвестного типа? Это может показаться безумным, но подумайте о том, как загружаются данные, — вы знаете, что в вашу игру поступает информация, но не знаете, какую форму она примет. В следующем подразделе мы узнаем, как с этим справиться.

Предполагаемые объявления

К счастью, C# может *предположить* тип переменной по присвоенному ей значению. Например, ключевое слово `var` может сообщить программе, что тип переменной `currentAge` должен быть определен по присвоенному ей значению `32`, которое является целым числом:

```
var currentAge = 32;
```

Иногда удобно так делать, но не поддавайтесь привычке ленивого программирования использовать предполагаемые объявления везде. Из-за этого в коде появится слишком много неясности там, где он должен быть кристально ясным.

Прежде чем мы завершим обсуждение типов данных и преобразования, нам нужно вкратце поговорить о создании пользовательских типов, что мы и сделаем.

Пользовательские типы

Когда мы говорим о типах данных, важно сразу понимать, что числа и слова (называемые *буквенными значениями*) — не единственные типы значений, которые может хранить переменная. Например, класс, структура или перечисление тоже могут быть сохранены как переменные. Мы начнем разговор об этом в главе 5 и продолжим в главе 10.

Итого про типы

Типы — сложная тема, и единственный способ освоить ее — использовать их. Однако следует помнить о некоторых важных вещах:

- все переменные должны быть определенного типа (будь он задан явно или через предположение);
- переменные могут содержать только значения назначенного им типа (в `int` нельзя «положить» текст);
- у каждого типа есть набор операций, которые он может и не может выполнять (`bool` нельзя вычесть из другого значения);

- если переменную объединить с переменной другого типа или присвоить значение другого типа, то необходимо выполнить преобразование (неявное или явное);
- компилятор C# может определить тип переменной по ее значению с помощью ключевого слова `var`, но этот метод следует использовать только в том случае, если тип заранее не известен.

Я рассказал множество мельчайших подробностей, упаковав их в несколько разделов, но и это еще не все. Вам по-прежнему нужно понять, как в C# устроены соглашения об именах, а также где находятся переменные в ваших скриптах.

Именование переменных

Выбор имен переменных может показаться второстепенным в свете всего, что вы узнали о модификаторах доступа и типах, но не все так просто. Четкие и последовательные соглашения об именах в вашем коде не только сделают его более читабельным, но и позволят другим разработчикам в команде понять ход ваших мыслей, не спрашивая вас.

Практические рекомендации

Первое правило именования переменных: выбираемое имя должно иметь смысл. Второе правило — используйте верблюжий регистр (camel case). Возьмем общий пример из игр и объявим переменную, в которой будет храниться здоровье игрока:

```
public int health = 100;
```

Вскоре после объявления такой переменной вам начнут поступать тревожные звонки от внутреннего голоса. Чье здоровье имеется в виду? Это максимальное или минимальное значение? На какой код повлияет изменение этого значения? На все эти вопросы легко ответить, выбрав осмысленное имя переменной. И нужно так и делать, если вы не хотите, чтобы ваш код запутал вас через неделю или месяц.

С учетом сказанного попробуем улучшить этот код, добавив заодно верблюжий регистр:

```
public int maxHealth = 100;
```

Помните, что в верблюжьем регистре имя переменной начинается со строчной буквы, а затем первая буква в каждом последующем слове становится заглавной. Таким образом проводится четкое различие между именами переменных и классов, которые начинаются с заглавной буквы.

Уже лучше. Немного подумав, мы обновили имя переменной с учетом ее смысла и контекста. Поскольку технических ограничений на длину имени переменной нет, вы можете переборщить и начать писать длинные описательные имена, но это будет так же плохо, как и короткое, неописательное имя.

Лучше всего делать имя переменной настолько информативным, насколько оно должно быть, — ни больше, ни меньше. Найдите свой стиль и придерживайтесь его.

Понятие области видимости переменных

Мы подошли к концу нашего разговора о переменных, но есть еще одна важная тема, которую нужно затронуть: область видимости. Подобно модификаторам доступа, определяющим, какие внешние классы могут получать информацию о переменной, область видимости переменной — это термин, используемый для описания того, где существует данная переменная и откуда она доступна в содержащем ее классе.

В C# есть три основных уровня области видимости переменных.

- Область видимости `global` выбирается для переменных, к которым может получить доступ вся программа (в данном случае игра). C# напрямую не поддерживает глобальные переменные, но эта концепция полезна в определенных случаях, которые мы рассмотрим в главе 10.

- Область видимости `class` или `member` выбирается для переменных, доступных в любом месте содержащего класса.
- Область видимости `local` выбирается для переменной, доступной только внутри определенного блока кода, в котором она создана.

Взгляните на рис. 3.6. Вам не нужно писать данный код в сценарий `LearningCurve`, если не хочется. Это всего лишь пример.

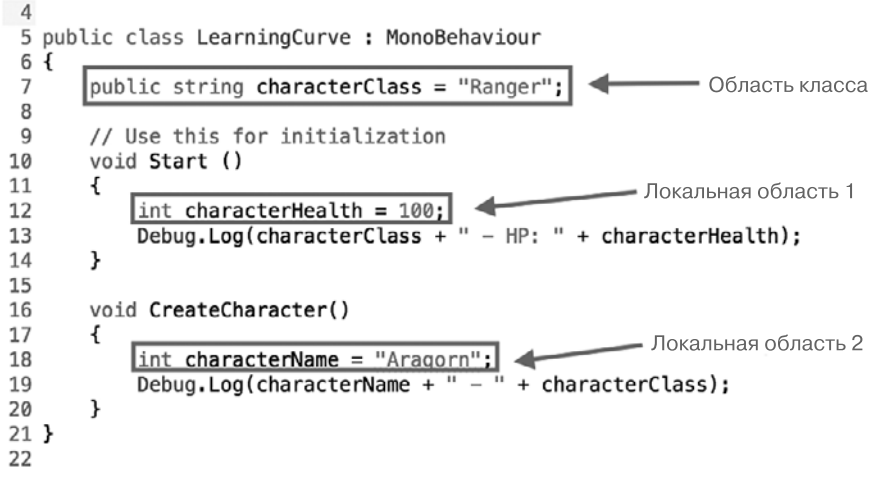


Рис. 3.6

Говоря о блоках кода, я имею в виду область внутри любого набора фигурных скобок. Они служат своего рода визуальной иерархией в программировании и чем дальше смещены вправо, тем глубже вложены в класс.

Более подробно поговорим об областях видимости переменных, показанных на рис. 3.6.

- Переменная `characterClass` объявлена в самом верху класса; это значит, что мы можем ссылаться на нее в любом месте сценария `LearningCurve`. Это иногда называют видимостью переменных — это действительно хорошая аналогия, которую стоит использовать.
- Переменная `characterHealth` объявляется внутри метода `Start()`; это значит, что она видна только внутри данного блока кода. Мы легко

можем получить доступ к `characterClass` из метода `Start()`, но если попытаемся получить доступ к `characterHealth` из другого места, то получим ошибку.

- Переменная `characterName` находится в тех же условиях, что и `characterHealth`, и доступ к ней можно получить только из метода `CreateCharacter()`. Таким образом, я показал, что в одном классе может быть несколько локальных областей (даже вложенных).

Проведя достаточно времени с программистами, вы наверняка услышите дискуссии (или споры, в зависимости от времени суток) о том, где лучше всего объявить переменную. Ответ проще, чем вы думаете: переменные следует объявлять с учетом их использования. Если у вас есть переменная, к которой нужно обращаться во всем классе, то сделайте ее переменной класса. Если она нужна только в определенном разделе кода, то объявите ее как локальную.

Обратите внимание: в окне Inspector можно просматривать только переменные класса, что не подходит для локальных или глобальных переменных.

Теперь, зная все об именах и областях видимости, вернемся за парту средней школы и заново узнаем, как работают арифметические операции!

Знакомство с операторами

Символы операторов в языках программирования выполняют *арифметические*, *присваивающие*, *реляционные* и *логические* функции, которые могут выполнять разные типы. Арифметические операторы представляют собой основные математические функции, а операторы присваивания одновременно выполняют математические функции и функции присваивания с заданным значением. Реляционные и логические операторы оценивают некое условие между несколькими значениями, например, *«больше»*, *«меньше»* или *«равно»*.

В C# также есть побитовые и прочие операторы, но они вам не пригодятся, пока вы не начнете создавать более сложные приложения.

На данный момент нам есть смысл рассматривать только арифметические операции и операторы присваивания, но мы перейдем к реляционным и логическим операциям в следующей главе, когда они потребуются.

Арифметика и присваивание

Вы с малых лет знакомы с символами арифметических операторов:

- `+` — сложение;
- `-` — вычитание;
- `/` — деление;
- `*` — умножение.

Операторы C# выполняются по обычному приоритету операций, то есть сначала вычисляется то, что в скобках, затем степени, умножение, деление, сложение и, наконец, вычитание[1]. Например, следующие равенства дадут разные результаты, несмотря на то что содержат одинаковые значения и операторы[2]:

```
5 + 4 - 3 / 2 * 1 = 8
5 + (4 - 3) / 2 * 1 = 5
```

В случае применения к переменным операторы работают так же, как и с чистыми значениями.

Операторы присваивания можно задействовать как сокращенную замену любой математической операции, используя символ арифметической операции и символ равенства вместе. Например, если бы мы

---

[1] Надо отметить, что в C# нет возведения в степень. Кроме того, автор в оригинале приводит аббревиатуру BEDMAS (Brackets, Exponents, Divide, Multiply, Add, Subtract), а потом в списке приоритетов приводит умножение раньше деления. Умножение и деление, как и сложение с вычитанием, являются операторами одного приоритета и выполняются по порядку слева направо. — *Примеч. науч. ред.*

[2] Тонкий момент: в обычной арифметике приведенные ниже выражения равны 7,5 и 5,5 соответственно, однако из-за того, что все операнды — целые числа, которые нигде явно не приведены к числам с плавающей точкой, используется целочисленная арифметика, в которой итоговое значение округляется по правилам коммерческого округления. — *Примеч. науч. ред.*

хотели умножить переменную на число, то оба следующих варианта дали бы один и тот же результат:

```
int currentAge = 32;
currentAge = currentAge * 2;
```

И второй вариант:

```
int currentAge = 32;
currentAge *= 2;
```

Символ равенства в C# также считается оператором присваивания. Другие символы присваивания следуют тому же синтаксическому шаблону, что и в предыдущем примере: +=, –= и /= выполняют сложение, вычитание, деление, а затем присваивание.

У строк операторы работают по-своему. Например, символ сложения у строк выполняет склейку:

```
string fullName = "Joe" + "Smith";
```

Такой подход обычно создает неуклюжий код, вследствие чего интерполяция строк становится предпочтительной для объединения различных фрагментов текста в большинстве случаев.

Теперь мы узнали, что у типов есть правила, которые определяют, какие операции и взаимодействия они могут иметь. Однако мы еще не попробовали это на практике и потому сделаем это прямо сейчас.

Время действовать. Выполняем операции на неправильных типах

Проведем небольшой эксперимент: попробуем перемножить строку и число с плавающей запятой, как делали ранее с числами (рис. 3.7).

Если вы посмотрите в окно консоли, то увидите сообщение об ошибке, которое гласит, что нельзя перемножать строку и число с плавающей запятой (рис. 3.8). Каждый раз, когда встречается подобная ошибка, необходимо проверить типы переменных на совместимость.

```
// Start is called before the first frame update
void Start()
{
    ComputeAge();

    Debug.Log($"A string can have variables like {firstName} inserted directly!");

    Debug.Log(firstName * pi);
}
```

Рис. 3.7

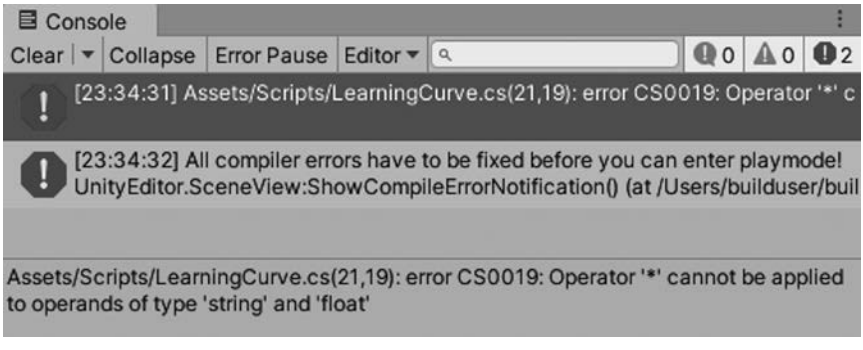


Рис. 3.8

Обязательно нужно убрать этот пример, поскольку компилятор не позволит нам запустить игру с такой ошибкой. Закомментируйте данную строку двумя чертами или просто удалите ее.

На данный момент это все, что нам нужно знать о переменных и типах. Обязательно проверьте себя, ответив на вопросы в конце главы, прежде чем двигаться дальше!

Определение методов

В предыдущей главе я кратко сказал о роли, которую методы играют в наших программах: они хранят и выполняют инструкции так же, как переменные хранят значения. Теперь вам нужно понять синтаксис объявлений методов и то, как они управляют действиями и поведением в классах.

Базовый синтаксис

Как и в случае с переменными, к объявлениям методов предъявляются следующие основные требования:

- тип данных, возвращаемый методом;
- уникальное имя (обычно начинается с заглавной буквы);
- пара скобок после имени метода;
- пара фигурных скобок, обозначающая тело метода (где хранятся инструкции).

Собирая все эти правила воедино, мы получаем простой шаблон метода:

```
returnType UniqueName()
{
    method body
}
```

В качестве практического примера разберем метод `Start()` из сценария `LearningCurve` (рис. 3.9).

```
13      // Use this for initialization
14      void Start ()
15      {
16          |
17      }
```

Рис. 3.9

Что здесь есть:

- метод начинается с ключевого слова `void`, которое задает возвращаемый тип метода, если он не возвращает никаких данных;
- у метода уникальное имя;
- после имени метода есть пара круглых скобок для возможных аргументов;
- тело метода ограничено набором фигурных скобок.

В общем, если у вас есть метод с пустым телом, то рекомендуется удалить его из класса. От неиспользуемого кода всегда стоит избавляться.

Как и у переменных, у методов есть уровни безопасности. А еще они могут иметь входные параметры, которые мы обсудим далее.

Модификаторы и параметры

У методов могут быть те же четыре модификатора доступа, что и у переменных, а также входные параметры. Параметры — это переменные-заполнители, которые могут передаваться в методы и быть доступными внутри них. Количество входных параметров, которые вы можете использовать, не ограничено, но каждый из них должен быть отделен запятой, для каждого указывается тип данных и уникальное имя.

Аргументы — это заполнители для переменных, значения которых можно использовать внутри тела метода.

Если мы применим эти параметры, то наш обновленный шаблон будет выглядеть так:

```
accessModifier returnType UniqueName(parameterType parameterName)
{
    method body
}
```

Если явного модификатора доступа нет, то по умолчанию используется private. Частный метод, как и частная переменная, не может быть вызван из других сценариев.

Чтобы вызвать метод (то есть запустить или выполнить его инструкции), мы просто используем его имя, за которым следует пара круглых скобок, с параметрами или без них и в конце ставим точку с запятой:

```
// Без параметров
UniqueName();

// С параметрами
UniqueName(parameterVariable);
```

Как и у переменных, каждый метод располагает информацией о его уровне доступа, типе возвращаемого значения и аргументах. Это называется сигнатурой метода. По сути, сигнатура помечает метод как уникальный для компилятора, поэтому Visual Studio знает, что с ним делать.

Теперь, когда мы понимаем, как устроены методы, создадим свой.

Время действовать. Определяем простой метод

Ранее, в предыдущей главе, вы слепо переписали метод `AddNumbers` в сценарий `LearningCurve`, не зная, во что ввязываетесь. На этот раз создадим метод осмысленно.

- Объявите метод с возвращаемым типом `void` и именем `GenerateCharacter()` как `public`.
- Добавьте оператор `Debug.Log()`, который выводит имя персонажа из вашей любимой игры или фильма.
- Вызовите метод `GenerateCharacter()` внутри метода `Start()` и нажмите кнопку Play (рис. 3.10).

```
// Use this for initialization
void Start()
{
    GenerateCharacter();
}

public void GenerateCharacter();
{
    Debug.Log("Character: Spike");
}
```

Рис. 3.10

При запуске игры Unity автоматически вызывает метод `Start()`, который, в свою очередь, вызывает метод `GenerateCharacter()` и выводит результат в окно консоли.

В различной документации можно встретить разную терминологию касаемо методов. В остальной части книги, когда мы будем создавать или объявлять метод, я буду называть это определением метода. Запуск или выполнение метода буду называть его вызовом.

Правильное именование — неотъемлемая часть программирования, поэтому неудивительно, что нам нужно посмотреть соглашения об именах для методов, прежде чем двигаться дальше.

Соглашения об именах

Подобно переменным, методам нужны уникальные и осмысленные имена, которые позволят различать их в коде. Методы определяют действия, поэтому рекомендуется называть их как действия. Например, метод `GenerateCharacter()` назван хорошо и понятно, а имя `Summary()` чересчур расплывчатое и не дает очень четкой картины того, что будет выполнять метод.

Названия методов всегда начинаются с заглавной буквы, и затем с заглавной буквы начинаются все последующие слова. Этот стиль называется *Pascal Case* (сводный брат формата *Camel Case*, который мы используем для переменных).

Методы — что-то вроде объездных дорог

Мы видели, что строки кода выполняются последовательно в том порядке, в котором они написаны, но использование методов сбивает эту последовательность. Вызов метода заставляет программу сначала выполнить по порядку его инструкции, а затем возобновить последовательное выполнение с того места, где был вызван метод.

Взгляните на рис. 3.11 и посмотрите, сможете ли понять, в каком порядке в консоль выведутся сообщения.

```
// Use this for initialization
void Start()
{
    Debug.Log("Choose a character.");
    GenerateCharacter();
    Debug.Log("A fine choice.");
}

public void GenerateCharacter();
{
    Debug.Log("Character: Spike");
}
```

Рис. 3.11

Порядок выполнения выглядит так.

1. Сообщение `Choose a Character` выводится первым, поскольку это первая строка кода.
2. Когда вызывается метод `GenerateCharacter()`, программа переходит к строке 23, выводит сообщение `Character: Spike`, а затем возобновляет выполнение со строки 17.
3. Сообщение `A fine choice` выводится последним, когда метод `GenerateCharacter()` завершит работу (рис. 3.12).

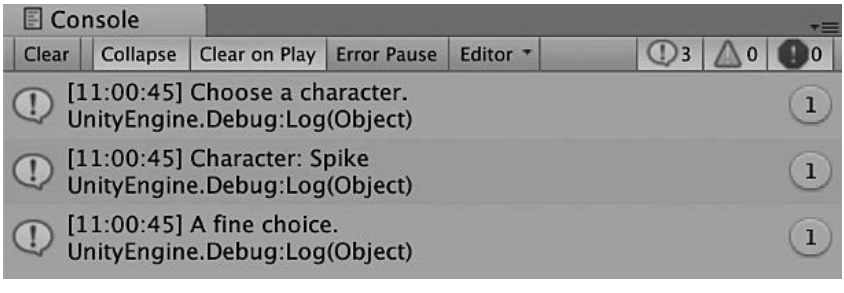


Рис. 3.12

Сами по себе методы были бы не очень полезными, кроме таких простых примеров, если бы мы не могли передавать им значения аргументов, что мы и сделаем.

Определение параметров

Вероятно, ваши методы не всегда будут такими же простыми, как `GenerateCharacter()`. Чтобы передать методу дополнительную информацию, следует определить список параметров, которые он будет принимать и использовать в работе. При определении параметра нужно задать две вещи:

- явный тип;
- уникальное имя.

Звучит знакомо? Параметры метода — это, по сути, упрощенные объявления переменных, и работают они так же. Каждый параметр действует как локальная переменная, доступная только внутри их конкретного метода.

Вы можете задавать столько параметров, сколько нужно. Независимо от того, пишете ли вы собственные методы или используете встроенные, у них будут параметры, которые требуются методу для выполнения указанной задачи.

Параметры — шаблон для типов значений, которые может принимать метод, а аргументы — это сами значения. Чтобы стало понятнее, уточню:

- аргумент, передаваемый в метод, должен соответствовать типу параметра, как переменная и ее значение;
- аргументы могут быть просто значениями (например, числом `2`) или переменными, объявленными в другом месте класса.

Имена аргументов и имена параметров не обязательно должны совпадать.

Теперь продолжим и добавим методу `GenerateCharacter()` параметры, чтобы код стал более интересным.

Время действовать. Добавляем параметры методу

Обновим метод `GenerateCharacter()`, чтобы он мог принимать два параметра.

1. Добавим два параметра: один — это имя персонажа строкового типа, а другой — уровень персонажа типа `int`.
2. В метод `Debug.Log()` добавим новые параметры.
3. Дополните вызов метода `GenerateCharacter()` в методе `Start()` своими аргументами, которые могут быть либо буквальными значениями, либо объявленными переменными (рис. 3.13).

Здесь мы определили два параметра, `name(string)` и `level (int)`, и задействовали их внутри метода `GenerateCharacter()`, словно это локальные переменные. Вызывая данный метод внутри `Start()`, мы указали значения аргументов для каждого параметра с соответствующими типами. На рис. 3.13 видно, что применение буквального строкового значения в кавычках дает тот же результат, что и использование переменной `characterLevel` (рис. 3.14).

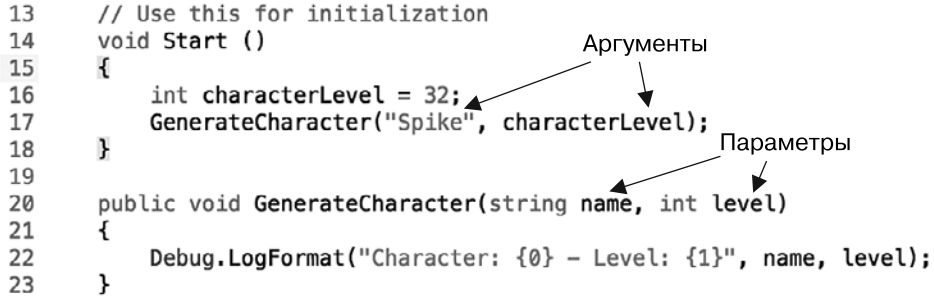


Рис. 3.13

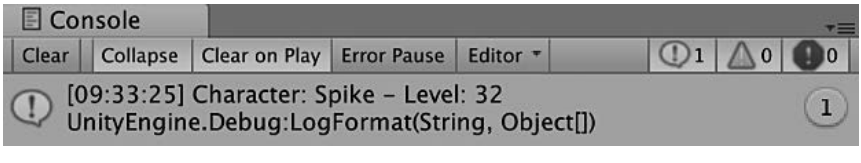


Рис. 3.14

Двигаемся дальше и узнаем, как передавать значения в метод и из него. Это подводит нас к следующему разделу — о возвращаемых значениях.

Определение возвращаемых значений

Помимо приема параметров, методы могут возвращать значения любого имеющегося в C# типа. Во всех наших предыдущих примерах использовался тип `void`, который ничего не возвращает, но главное

преимущество методов — это возможность выполнять вычисления и возвращать обратно вычисленные результаты.

Согласно шаблону возвращаемые типы методов указываются после модификатора доступа. Помимо типа, метод должен содержать ключевое слово `return`, за которым следует возвращаемое значение. Оно может быть переменной, буквальным значением или даже выражением, если соответствует объявленному возвращаемому типу.

Методы, у которых тип возвращаемого значения задан как void, по-прежнему позволяют использовать ключевое слово return без присвоенного значения или выражения. Как только будет достигнута строка с ключевым словом return, метод прекратит выполнение. Это полезно в тех случаях, когда вы хотите избежать определенного поведения или защититься от сбоев программы.

Добавим тип возвращаемого значения в `GenerateCharacter()` и узнаем, как записать его в переменную.

Время действовать. Добавляем возвращаемый тип

Научим метод `GenerateCharacter` возвращать целое число.

1. Заменим тип возвращаемого значения в объявлении метода с `void` на `int`.
2. Вернем значение, увеличенное на +5, используя ключевое слово `return` (рис. 3.15).

```
20      public int GenerateCharacter(string name, int level)
21      {
22          Debug.LogFormat("Character: {0} - Level: {1}", name, level);
23          return level + 5;
24      }
```

Рис. 3.15

Теперь метод `GenerateCharacter()` возвращает целое число. Оно вычисляется путем прибавления `5` к аргументу `level`. Но мы пока не указали,

как хотим использовать это возвращаемое значение, из чего следует, что прямо сейчас сценарий не будет делать ничего нового.

Теперь возникает вопрос: как нам получить и использовать вновь добавленное возвращаемое значение? Об этом — в следующем подразделе.

Использование возвращаемых значений

В этом вопросе есть два подхода.

- Создать локальную переменную для захвата (хранения) возвращаемого значения.
- Задействовать сам вызывающий метод как замену возвращаемому значению, используя его как переменную. Вызывающий метод — это фактическая строка кода, запускающая инструкции, которыми в нашем примере будет `GenerateCharacter("Spike", characterLevel)`. При необходимости даже можно передать вызывающий метод другому методу в качестве аргумента.

В большинстве случаев программисты предпочитают первый вариант из-за его удобочитаемости. Указание вызовов методов в качестве переменных может породить беспорядок в коде, особенно когда мы используем одни методы в качестве аргументов в других методах.

Попробуем в коде захватить и вывести возвращаемое значение метода `GenerateCharacter()`.

Время действовать. Захватываем возвращаемые значения

Мы собираемся опробовать оба способа захвата и использования возвращаемых переменных.

1. Создадим новую локальную переменную типа `int` с именем `nextSkillLevel` и присвоим ей возвращаемое значение вызова метода `GenerateCharacter()`, который у нас уже есть.

2. Добавим два оператора вывода, первый из которых выводит `nextSkillLevel`, а второй — новый вызывающий метод со значениями аргументов по вашему выбору.
3. Закомментируйте вывод внутри `GenerateCharacter()` двумя косыми чертами (`//`), чтобы в консоли не было нагромождений.
4. Сохраните файл и нажмите кнопку Play в Unity (рис. 3.16).

```
13      // Use this for initialization
14      void Start ()
15      {
16          int characterLevel = 32;
17
18          int nextSkillLevel = GenerateCharacter("Spike", characterLevel);
19          Debug.Log(nextSkillLevel);
20          Debug.Log(GenerateCharacter("Faye", characterLevel));
21      }
22
23      public int GenerateCharacter(string name, int level)
24      {
25          //Debug.LogFormat("Character: {0} - Level: {1}", name, level);
26          return level + 5;
27      }
```

Рис. 3.16

Для компилятора переменная `nextSkillLevel` и вызывающий метод `GenerateCharacter()` представляют одну и ту же информацию, а именно целое число, поэтому в обоих выводах появляется число `37` (рис. 3.17).

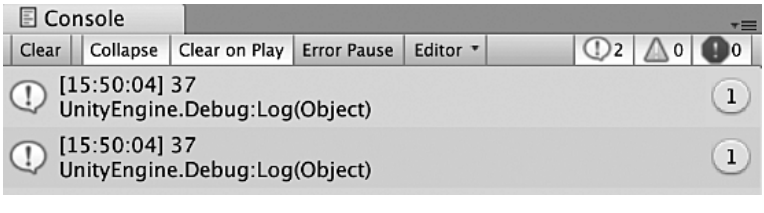


Рис. 3.17

Мы обсудили много всего, особенно учитывая огромные возможности методов при работе с параметрами и возвращаемыми значениями. Однако здесь мы на минуту притормозим и рассмотрим некото-

рые из наиболее распространенных методов Unity, чтобы немного передохнуть.

Испытание героя. Рассматриваем методы как аргументы

Если не боитесь, то почему бы не попробовать создать новый метод, который принимает параметр типа `int` и просто выводит его в консоль? Возвращаемый тип не требуется. Создав этот метод, вызовите его в `Start()`, передайте вызов метода `GenerateCharacter` в качестве аргумента и посмотрите на результат.

Анализ распространенных методов Unity

Теперь мы можем обсудить наиболее распространенные встроенные методы, появляющиеся в новых сценариях C# в Unity: `Start()` и `Update()`. В отличие от методов, которые мы определяем сами, методы, принадлежащие классу `MonoBehaviour`, автоматически вызываются движком Unity по особым правилам. В большинстве случаев важно иметь в сценарии хотя бы один метод `MonoBehaviour` для запуска вашего кода.

Вы можете найти полный список всех доступных методов MonoBehaviour и их описания, пройдя по ссылке docs.unity3d.com/ScriptReference/MonoBehaviour.html.

Рассказывая историю, всегда лучше начинать с самого начала. Так и здесь — начнем с встроенного в Unity метода `Start()`.

Метод Start()

Unity вызывает этот метод в первом кадре, в котором включается сценарий. Поскольку сценарии `MonoBehaviour` почти всегда прикреплены к объектам `GameObject` на сцене, прикрепленные к ним сценарии

включаются одновременно с их загрузкой при нажатии кнопки Play. В нашем проекте сценарий `LearningCurve` присоединен к объекту `Main Camera`; это значит, что его метод `Start()` запускается, когда камера загружается в сцену. Метод `Start()` в основном используется для установки переменных или выполнения логики, которая должна произойти перед первым запуском метода `Update()`.

Все примеры, которые мы рассматривали до сих пор, работали в методе Start(), даже если не выполняли действия по настройке, но обычно так не делается. Однако он срабатывает только один раз, что делает его отличным инструментом для вывода одноразовой информации в консоли.

Помимо метода `Start()`, есть еще один важный метод Unity, с которым вам придется работать постоянно: метод `Update()`. Вы познакомитесь с ним в следующем подразделе, а затем мы закончим эту главу.

Метод Update()

Если вы изучите пример кода, пройдя по ссылке docs.unity3d.com/ScriptReference/, то заметите, что большая часть кода выполняется в методе `Update()`. Во время работы игры окно сцены отрисовывается много раз в секунду. Количество отрисовок определяется параметром FPS — Frames Per Second, или количество кадров в секунду. После отображения каждого кадра вызывается метод `Update()`, вследствие чего он становится одним из наиболее часто выполняемых методов в игре. Это делает его идеальным для перехвата ввода с помощью мыши и клавиатуры или выполнения логики игрового процесса.

Если вам интересно узнать FPS на вашем компьютере, то нажмите кнопку Play и кнопку Stats в правом верхнем углу окна Game (рис. 3.18).

Вы будете использовать методы `Start()` и `Update()` в большей части ваших сценариев C#, поэтому ознакомьтесь с ними. Мы подошли к концу данной главы, располагая полным набором знаний об основных элементах программирования.

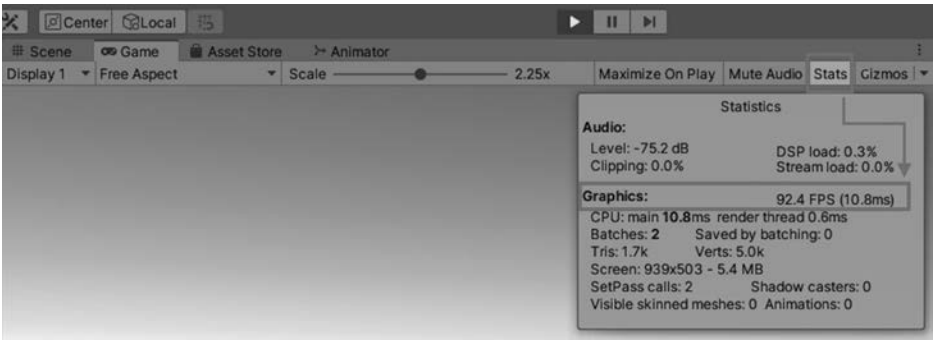


Рис. 3.18

Подведем итоги

В этой главе мы стремительно перешли от базовой теории программирования и ее основных элементов к реальному коду и синтаксису C#. Вы видели хорошие и плохие варианты форматирования кода, узнали, как отлаживать информацию в консоли Unity, и создали ваши первые переменные. Вдобавок познакомились с типами C#, модификаторами доступа и областью видимости переменных, поскольку работали с переменными на панели Inspector и начали углубляться в методы и действия.

Методы помогли вам понять письменные инструкции в коде, но, что еще более важно, вы узнали, как правильно использовать их возможности, чтобы получить нужное поведение. Входные параметры, типы возвращаемых данных и сигнатуры методов — все это важные темы, но самое главное — это возможность выполнения новых видов действий. Теперь вы вооружены двумя фундаментальными элементами программирования, и почти все действия, которые вам предстоит совершить далее, будут расширять или применять эти две концепции.

В следующей главе мы рассмотрим специальное подмножество типов C#, называемых коллекциями, которые позволяют хранить группы связанных данных, вы узнаете, как писать код, основанный на решениях.

Контрольные вопросы. Переменные и методы

1. Как правильно задавать имя переменной на C#?
2. Как сделать так, чтобы переменная появилась на панели Inspector в Unity?
3. Какие четыре модификатора доступа есть в C#?
4. Когда нужны явные преобразования между типами?
5. Что нужно для определения метода?
6. Для чего нужны круглые скобки в конце имени метода?
7. Что означает возвращаемый тип `void` в определении метода?
8. Как часто Unity вызывает метод `Update()`?

4 Поток управления и типы коллекций

Одна из главных задач компьютера — следить за тем, что происходит при каких-то заранее известных условиях. Щелкая на папке, вы ожидаете, что она откроется. Печатая на клавиатуре, вы ожидаете, что на экране будет появляться соответствующий текст.

Написание кода для приложений или игр концептуально такое же. Ваше приложение должно в определенном состоянии вести себя определенным образом, а при изменении условий менять поведение. В терминах программирования это называется потоком управления или, другими словами, потоком выполнения кода в различных сценариях.

Помимо работы с управляющими операторами, в текущей главе мы на практике рассмотрим типы данных под названием *«коллекции»*. Это такие типы данных, которые позволяют хранить несколько значений или группы значений в одной переменной. Они часто применяются совместно с часто встречающимися сценариями потока управления.

В свете вышесказанного мы обсудим такие темы, как:

- операторы выбора;
- работа с коллекциями типа `Array`, `Dictonary` и `List`;
- операторы итерации: циклы `for`, `foreach` и `while`;
- управление выполнением с помощью ключевых слов `break`, `continue` и `return`;
- как не допустить появления бесконечных циклов.

Операторы выбора

Самые сложные задачи программирования часто можно свести к некоему набору вариантов действий, которые оценивает и выполняет игра или программа. Поскольку Visual Studio и Unity не могут выбрать за вас, вы должны сами прописать логику принятия решения.

Операторы выбора `if-else` и `switch` позволяют задать разветвление программы на основе одного или нескольких условий, а также действия, которые вы хотите предпринять в каждом из описанных случаев. Обычно эти условия берутся из:

- ввода со стороны пользователя;
- оценки выражений и логики;
- сравнения переменных или значений.

Мы начнем с простейшего из этих условных операторов: `if-else`.

Оператор if-else

Оператор `if-else` — наиболее распространенный способ принятия решений в коде. Если не брать пока синтаксис, основная идея состоит в следующем: *если некое условие выполнено, то мы должны выполнить один блок кода, а если нет, то другой*. Данный оператор сродни множеству дверей, а условие — это ключ от одной из них. Чтобы пройти, ключ должен подходить к замку. В противном случае во входе будет отказано и код будет пробовать другие «двери». Посмотрим, как выглядит синтаксис этого оператора.

Базовый синтаксис

Чтобы записать оператор `if-else`, нам потребуется:

- ключевое слово `if` в начале строки;
- пара круглых скобок, куда мы запишем условие;
- тело условия:

```
if (условие истинно)
    Выполнить эту строку кода
```

Однако если тело оператора состоит более чем из одной строки, то потребуется также пара фигурных скобок, в которых можно будет записать больший блок кода:

```
if (условие истинно)
{
    Выполнить эти строки
    кода
}
```

При желании мы можем добавить оператор `else`, чтобы перечислить действия, которые должны произойти, если условие окажется ложным. Оформление будет таким же:

```
else
    Выполнить эту строку кода

// ИЛИ

else
{
    Выполнить эти строки
    кода
}
```

Если собрать все воедино, то синтаксис оператора будет читаться почти как предложение:

```
if(условие истинно)
{
    Выполнить эти строки
    кода
}
else
{
    Выполнить эти строки
    кода
}
```

Данный оператор дает отличный повод потренировать логическое мышление, по крайней мере в программировании, поэтому разберем три различных варианта `if-else` более подробно.

1. Единственный оператор `if` может существовать сам по себе в тех случаях, когда вас не волнует, что произойдет при невыполнении условия. В следующем примере, если переменная `hasDungeonKey` имеет значение `true`, будет выведено сообщение, а если `false`, то ничего не произойдет (рис. 4.1).

```
public class LearningCurve : MonoBehaviour
{
    public bool hasDungeonKey = true;

    // Use this for initialization
    void Start()
    {
        if(hasDungeonKey)
        {
            Debug.Log("You possess the sacred key - enter.");
        }
    }
}
```

Рис. 4.1

Если условие «выполнено», то имеется в виду, что его результат равен true, поэтому иногда такое условие еще называют «пройденным».

2. Оператор `else` мы добавляем в случаях, когда необходимо выполнить что-то и в случае истинности, и в случае ложности условия. Если переменная `hasDungeonKey` окажется ложной, то оператор `if` пропустит свое тело и передаст выполнение кода оператору `else` (рис. 4.2).

```
public class LearningCurve : MonoBehaviour
{
    public bool hasDungeonKey = true;

    // Use this for initialization
    void Start()
    {
        if(hasDungeonKey)
        {
            Debug.Log("You possess the sacred key - enter.");
        }
        else
        {
            Debug.Log("You have not proved yourself worthy, warrior.");
        }
    }
}
```

Рис. 4.2

3. Если нужно рассмотреть несколько возможных вариантов, то можно добавить оператор `else-if` с такими же круглыми скобками, условиями и фигурными скобками. Лучше покажу сразу на примере.

Имейте в виду, что только оператор if может использоваться сам по себе, в отличие от остальных рассмотренных нами операторов.

Вы можете создавать и более сложные условия с помощью основных математических операций, таких как > (больше), < (меньше), >= (больше или равно), <= (меньше или равно), и == (равно).

Например, условие (2>3) вернет false и не пройдет, а условие (2<3) вернет true и пройдет.

Но пока не беспокойтесь ни о чем — мы скоро рассмотрим все на примерах.

Время действовать. Программируем карманника

Напишем оператор `if-else`, который проверяет количество денег в кармане персонажа, возвращая разные журналы отладки для трех разных случаев: больше 50, меньше 15 и все остальное.

1. Откройте сценарий `LearningCurve` и добавьте новую переменную типа `int` с именем `currentGold`. Задайте ей значение от 1 до 100.
2. Добавьте оператор `if`, который будет проверять, превышает ли значение переменной `currentGold` число 50, и выведите сообщение в консоль, если это так.
3. Добавьте оператор `else-if`, чтобы сделать то же самое для значения меньше 15.
4. Добавьте оператор `else` без условий и какой-нибудь вывод в консоль по умолчанию.
5. Сохраните файл и нажмите кнопку Play (рис. 4.3).

```
public class LearningCurve : MonoBehaviour
{
    public int currentGold = 32;

    // Use this for initialization
    void Start()
    {
        if(currentGold > 50)
        {
            Debug.Log("You're rolling in it - beware of pickpockets.");
        }
        else if (currentGold < 15)
        {
            Debug.Log("Not much there to steal.");
        }
        else
        {
            Debug.Log("Looks like your purse is in the sweet spot.");
        }
    }
}
```

Рис. 4.3

Если в моем примере мы зададим переменной `currentGold` значение `32`, то код выполнится следующим образом.

1. Оператор `if` и вывод в консоль пропускаются, поскольку значение `currentGold` не превышает 50.
2. Оператор `else-if` и вывод в консоль также пропускаются, поскольку `currentGold` не меньше 15.
3. Поскольку ни одно из предыдущих условий не было выполнено, выполняется оператор `else` и выводится третье сообщение (рис. 4.4).

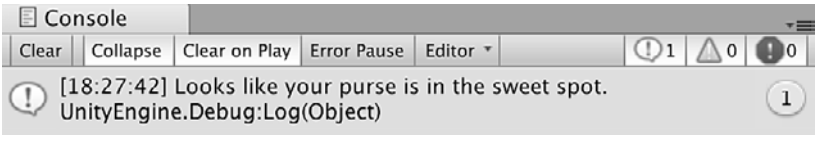


Рис. 4.4

Самостоятельно попробуйте задавать другие значения переменной `currentGold` и увидите, что произойдет в разных случаях.

Использование оператора NOT

В реальных задачах не всегда требуется проверка условия именно на истинность. И если это так, то нам поможет оператор `NOT`. Будучи написанным с помощью восклицательного знака, этот оператор допускает выполнение отрицательных или ложных условий с помощью оператора `if` или `else-if`. То есть приведенные ниже условия с точки зрения логики одинаковы:

```
if(variable == false)

// И

if(!variable)
```

Как вы уже знаете, в условии `if` можно проверять логические значения, буквальные значения или выражения. Естественно, оператор `NOT` должен работать во всех этих случаях. Рассмотрим пример, где в операторе `if` проверяется два разных ложных значения, `hasDungeonKey` и `WeaponType` (рис. 4.5).

```
5 public class LearningCurve : MonoBehaviour
6 {
7     public bool hasDungeonKey = false;
8     public string weaponType = "Arcane Staff";
9
10    // Use this for initialization
11    void Start()
12    {
13        if(!hasDungeonKey)
14        {
15            Debug.Log("You may not enter without the sacred key.");
16        }
17
18        if(weaponType != "Longsword")
19        {
20            Debug.Log("You don't appear to have the right type of weapon...");
21        }
22    }
23 }
```

Рис. 4.5

Эти операторы работают следующим образом.

- Первый оператор читается так: «Если значение переменной `hasDungeonKey` равно `false`, то оператор `if` принимает значение `true` и выполняет свой блок кода».

Если вам непонятно, как ложное значение может вдруг превратиться в истинное, то подумайте об этом так: оператор if проверяет истинность не значения, а всего выражения в целом. Переменная hasDungeonKey может быть равна false, но оператор проверяет не ее, а результат всего выражения.

- Второй оператор можно перевести как «Если строковое значение `WeaponType` не равно `Longsword`, то выполнить этот блок кода».

Результаты отладки приведены на рис. 4.6. Однако если вам все еще что-то непонятно, то скопируйте этот код в сценарий `LearningCurve` и поиграйте со значениями переменных, пока не станет ясно.

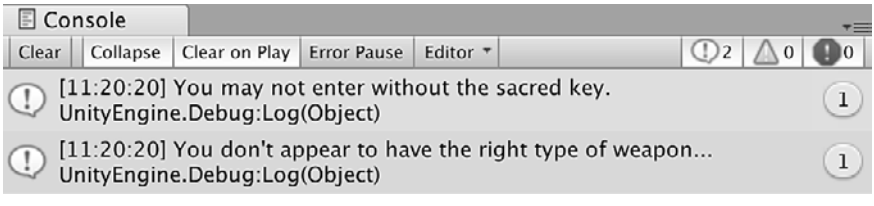


Рис. 4.6

До сих пор наши операторы ветвления были довольно простыми, но C# также позволяет вкладывать условные операторы друг в друга, чтобы описывать более сложные ситуации.

Вложенные условия

Одна из наиболее ценных функций операторов `if-else` заключается в том, что они могут быть вложены друг в друга, вследствие чего в коде можно описывать достаточно сложную логику. В программировании это называется деревом решений. Как и в настоящем коридоре, за дверями могут быть другие двери, и все это вместе создает лабиринт возможностей (рис. 4.7).

Разберем представленный пример.

- Первый оператор `if` проверяет значение переменной `WeaponEquipped`. На данный момент код проверяет лишь сам факт экипировки и не смотрит, какое именно это оружие.
- Второй оператор `if` проверяет переменную `WeaponType` и выводит соответствующее сообщение в консоль.

- Если первый оператор `if` имеет значение `false`, то код перейдет к оператору `else` и выполнит его. Если второй оператор `if` тоже будет иметь значение `false`, то ничего не выведется, поскольку блока `else` у него нет.

```
public class LearningCurve : MonoBehaviour
{
    public bool weaponEquipped = true;
    public string weaponType = "Longsword";

    // Use this for initialization
    void Start()
    {
        if(weaponEquipped)
        {
            if(weaponType == "Longsword")
            {
                Debug.Log("For the Queen!");
            }
        }
        else
        {
            Debug.Log("Fists aren't going to work against armor...");
        }
    }
}
```

Рис. 4.7

Продумывание логических результатов полностью ложится на программиста. Вам решать, какие ветви будут в коде и какими будут результаты выполнения.

Полученные в этом подразделе знания помогут вам без проблем разобраться в простых задачах. Тем не менее очень скоро вам потребуются более сложные инструкции, и здесь на сцену выходят сложные условия.

Сложные условия

Помимо вложенных операторов, вы можете объединять несколько проверок условий в один оператор `if` или `else-if` с логическими операторами `AND` и `OR`:

- оператор `AND` записывается двумя символами амперсанда, `&&`. Сложное условие с оператором `AND` будет истинно в случае, если все входящие в него условия будут истинны;

- оператор OR записывается двумя вертикальными чертами, ||. Сложное условие с оператором OR будет истинно в случае, если хотя бы одно входящее в него условие будет истинно.

В следующем примере (рис. 4.8) представлен новый оператор if, который теперь проверяет и WeaponEquipped, и WeaponType, при этом для выполнения блока кода оба условия должны быть истинными.

```
13      if{WeaponEquipped && WeaponType == "Longsword"}
14      {
15          Debug.Log("For the Queen!");
16      }
```

Рис. 4.8

Операторы AND и OR можно комбинировать в целях проверки множества условий в любом порядке. В языке нет ограничений на количество операторов, которые вы можете комбинировать. Но будьте аккуратны при написании сложных выражений и не создавайте логических условий, которые никогда не будут выполняться.

Пришло время собрать воедино все, что вы узнали об операторах if. Повторите этот подраздел, если нужно, а затем переходите к следующему.

Время действовать. Ищем сокровища

Закрепим эту тему небольшим экспериментом с сундуком с сокровищами.

1. Объявим три переменные в верхней части сценария LearningCurve: pureOfHeart — логическое значение, которое должно быть истинным; hasSecretIncantation — тоже логическое значение, которое будет false; а rareItem — строка, и ее значение вы зададите сами.
2. Создайте публичный метод без возвращаемого значения с именем OpenTreasureChamber и вызовите его внутри метода Start.
3. Внутри метода OpenTreasureChamber объявите оператор if-else, в котором проверьте, истинно ли значение pureOfHeart и совпадает ли rareItem со строковым значением, которое вы ему присвоили.

4. Создайте вложенный оператор `if-else` внутри первого, который проверит, является ли переменная `hasSecretIncantation` ложной.
5. В каждом варианте оператора `if-else` выведите что-нибудь в консоль, сохраните сценарий и нажмите кнопку Play (рис. 4.9).

```
public class LearningCurve : MonoBehaviour
{
    public bool pureOfHeart = true;
    public bool hasSecretIncantation = false;
    public string rareItem = "Relic Stone";

    // Use this for initialization
    void Start()
    {
        OpenTreasureChamber();
    }

    public void OpenTreasureChamber()
    {
        if (pureOfHeart && rareItem == "Relic Stone")
        {
            if(!hasSecretIncantation)
            {
                Debug.Log("You have the spirit, but not the knowledge.");
            }
            else
            {
                Debug.Log("The treasure is yours, worthy hero!");
            }
        }
        else
        {
            Debug.Log("Come back when you have what it takes.");
        }
    }
}
```

Рис. 4.9

Если вы напишете свой код так же, как показано на этом рисунке, то будет выведено сообщение из оператора `if`. Это значит, что наш код прошел первую проверку оператора `if` на два условия, но не сработал в третьем (рис. 4.10).

Теперь вы можете приостановиться и попробовать задействовать еще более крупные операторы `if-else` в своих задачах, но это не будет

эффективным в долгосрочной перспективе. Хорошее программирование подразумевает использование правильного инструмента для каждой задачи, и здесь на помощь приходит оператор `switch`.

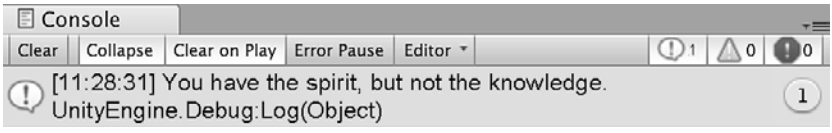


Рис. 4.10

Оператор switch

Операторы `if-else` отлично позволяют описывать логику принятия решений. Однако при наличии более трех или четырех вариантов ветвления использовать эти операторы просто невозможно. И пока вы это поймете, ваш код может в конечном итоге превратиться в запутанный клубок, за которым трудно уследить, а его обновление станет вызывать головную боль. Операторы `switch` принимают выражения и позволяют записывать действия для каждого возможного результата, но в гораздо более кратком формате, чем `if-else`.

Базовый синтаксис

Для операторов `switch` требуются следующие элементы:

- ключевое слово `switch`, за которым следует пара круглых скобок с условием;
- пара фигурных скобок;
- оператор `case` для каждого возможного варианта с двоеточием в конце:
  - отдельные строки кода или методы, за которыми следует ключевое слово `break` и точка с запятой;
- оператор `default` для вариантов по умолчанию, заканчивающийся двоеточием:
  - отдельные строки кода или методы, за которыми следует ключевое слово `break` и точка с запятой.

Описанный шаблон выглядит так:

```
switch(matchExpression)
{
    case matchValue1:
        Выполнить эти строки кода
        break;
    case matchValue2:
        Выполнить эти строки кода
        break;
    default:
        Выполнить эти строки кода
        break;
}
```

Выделенные ключевые слова здесь самые важные. Когда определен оператор `case`, все, что находится между его двоеточием и ключевым словом `break`, работает так же, как блок кода оператора `if-else`. Ключевое слово `break` дает программе указание полностью выйти из оператора `switch` после срабатывания выбранного `case`. Теперь обсудим, как оператор определяет, какой именно `case` выполняется. Это называется сопоставлением с образцом.

Сопоставление с образцом

В операторах `switch` сопоставлением с образцом называется проверка выражения на соответствие нескольким операторам `case`. Выражение сопоставления может быть любого типа, кроме `null`. Все значения оператора `case` должны по типу соответствовать выражению сопоставления.

Например, если бы у нас был оператор `switch`, который оценивает целочисленную переменную, то каждый оператор `case` должен был бы содержать целочисленное значение для сравнения. В каком операторе `case` окажется значение, соответствующее выражению, тот в итоге и выполнится. Если ни один `case` не подходит, то срабатывает `default`. Проверим!

Время действовать. Выбираем действие

Мы рассмотрели много нового синтаксиса и информации, но всегда лучше посмотреть на примере. Создадим простой оператор `switch` для различных действий, которые может выполнять персонаж.

1. Создайте новую строковую переменную (в классе или локальную) с именем `characterAction` и установите для нее значение `Attack`.
2. Объявите оператор `switch` и используйте `characterAction` в качестве выражения соответствия.
3. Создайте два оператора `case` для `Heal` и `Attack` с разными выводами в консоль. Не забудьте добавить ключевое слово `break` в конце каждого из них.
4. Добавьте `default` и вывод в консоль для него.
5. Готово! Сохраните файл и нажмите кнопку `Play` в Unity (рис. 4.11).

```
5 public class LearningCurve : MonoBehaviour
6 {
7     // Use this for initialization
8     void Start()
9     {
10        string characterAction = "Attack";
11
12        switch(characterAction)
13        {
14            case "Heal":
15                Debug.Log("Potion sent.");
16                break;
17            case "Attack":
18                Debug.Log("To arms!");
19                break;
20            default:
21                Debug.Log("Shields up.");
22                break;
23        }
24    }
25 }
```

Рис. 4.11

Поскольку для переменной `characterAction` установлено значение `Attack`, оператор `switch` выполняет второй вариант и выводит сообщение. Измените `characterAction` либо на `Heal`, либо на неопределенное действие, чтобы проверить первый случай и случай по умолчанию (рис. 4.12).

Бывают случаи, когда к выполнению одного и того же действия должны приводить несколько операторов `case`, но не все. Это так называемые «провалы», и о них поговорим далее.

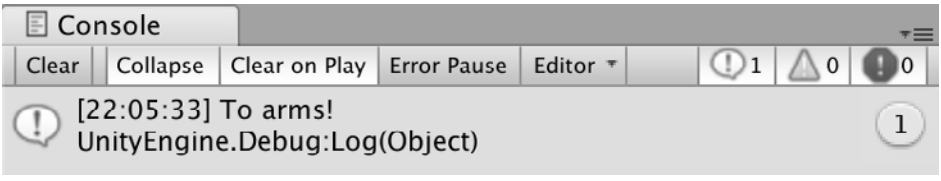


Рис. 4.12

«Провалы»

Операторы `Switch` могут выполнять одно и то же действие для нескольких `case`, как ранее мы задавали несколько условий в одном операторе `if`. Это называется «провалом». Если блок `case` оставлен пустым или в нем записан код без ключевого слова `break`, то он перейдет в `case` непосредственно под ним.

Операторы case можно писать в любом порядке, поэтому создание таких «провалов» значительно повышает читабельность и эффективность кода.

Добавим код в метод `SwitchingAround()`, чтобы посмотреть на него в действии.

Время действовать. Бросаем кубики

Смоделируем сценарий настольной игры с оператором `switch` и случаем «провала», когда бросок игральных кубиков определяет будущее действие.

1. Создайте переменную типа `int` с именем `diceRoll` и присвойте ей значение `7`.
2. Объявите оператор `switch` с `diceRoll` в качестве выражения соответствия.
3. Добавьте три оператора `case` для возможных бросков кубиков: `7`, `15` и `20`.
4. Вариантам `15` и `20` назначьте свои журналы отладки и операторы прерывания, а случай `7` должен переходить в случай `15`.
5. Сохраните файл и запустите его в Unity (рис. 4.13).

```
7     // Use this for initialization
8     void Start()
9     {
10        int diceRoll = 7;
11
12        switch(diceRoll)
13        {
14            case 7:
15            case 15:
16                Debug.Log("Mediocre damage, not bad.");
17                break;
18            case 20:
19                Debug.Log("Critical hit, the creature goes down!");
20                break;
21            default:
22                Debug.Log("You completely missed and fell on your face.");
23                break;
24        }
25    }
```

Рис. 4.13

Если вы хотите посмотреть, как сработает «провал», то попробуйте добавить вывод сообщения в case 7, но без ключевого слова break.

Если значение переменной `diceRoll` равно `7`, то оператор `switch` попадет в первый `case`, который «провалится» и выполнит вариант `15`, поскольку в нем отсутствуют блок кода и оператор `break`. Если вы измените `diceRoll` на `15` или `20`, то в консоли выведутся соответствующие сообщения, а любое другое значение выполнит `default` в конце (рис. 4.14).

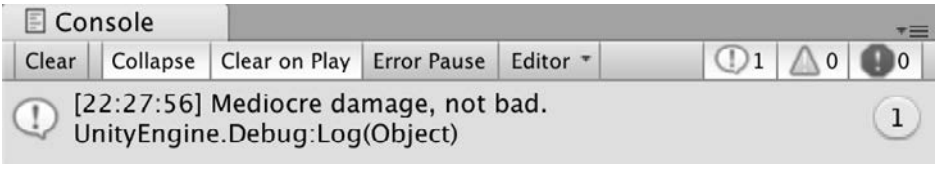


Рис. 4.14

Операторы switch чрезвычайно эффективны и позволяют просто изложить даже самую сложную логику принятия решений. Если вы хотите глубже изучить их работу, то перейдите по ссылке docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/switch.

На данный момент это все, что нам нужно знать об условной логике. Итак, повторите данный раздел, если нужно, а затем проверьте себя, ответив на контрольные вопросы ниже, прежде чем переходить к коллекциям.

Контрольные вопросы 1. Операторы if, and и or

Проверьте свои знания, ответив на следующие вопросы.

1. Какие значения используются для проверки операторов `if`?
2. Какой оператор может превратить истинное условие в ложное или ложное в истинное?
3. Если для выполнения кода оператора `if` должны выполняться сразу два условия, то какой логический оператор вы бы использовали для их объединения?
4. Если для выполнения кода оператора `if` достаточно выполнения только одно из двух условий, то какой логический оператор вы бы использовали для соединения этих двух условий?

Теперь мы готовы перейти к изучению коллекций. Эти типы откроют перед нами новый мир программирования игр и программ на C#!

Знакомство с коллекциями

Пока нам нужны были только переменные, хранящие одно значение, но есть много случаев, в которых требуется множество значений. В C# есть несколько типов коллекций: массивы, словари и списки. У каждого из этих типов есть свои сильные и слабые стороны, которые мы обсудим ниже.

Массивы

Массивы — самая базовая коллекция в C#. Массив — это контейнер для группы значений, называемых *элементами массива*, к каждому из которых можно обращаться и задавать значение по отдельности:

- массивы могут хранить значения любого типа, все элементы должны быть одного типа;
- длина массива или количество элементов, которые содержит массив, задаются в момент создания массива и не могут быть изменены в дальнейшем;
- если при создании массива не задать начальные значения, то каждому элементу будет присвоено значение по умолчанию. Элементы в числовых массивах по умолчанию равны нулю, а в массивах любого другого типа устанавливается значение `null` (то есть отсутствие значения).

Массивы — наименее гибкий тип коллекций в C#. В основном это связано с тем, что его элементы нельзя добавлять или удалять после создания. Зато массивы позволяют хранить информацию, которая, скорее всего, не изменится. Отсутствие гибкости делает их более быстрыми по сравнению с другими типами коллекций.

Базовый синтаксис

Объявление массива похоже на другие типы переменных, с которыми мы работали, но есть и отличия:

- для массива требуется указать тип элементов, пару квадратных скобок и уникальное имя;
- затем используется ключевое слово `new` для создания массива в памяти, после него указывается тип значения и еще одна пара квадратных скобок;
- количество элементов, которые хранятся в массиве, указывается во второй паре квадратных скобок.

Шаблон объявления массива выглядит так:

```
elementType[] name = new elementType[numberOfElements];
```

Рассмотрим пример, в котором нам нужно сохранить три лучших результата в некой игре:

```
int[] topPlayerScores = new int[3];
```

То есть `topPlayerScores` — это массив целых чисел, в котором будут храниться три элемента. Поскольку мы не добавляли никаких начальных значений, все три значения в `topPlayerScores` будут равны `0`.

Вы можете присвоить значения элементам массива в момент его создания, указав их в фигурных скобках в конце объявления переменной. В C# есть как длинные, так и сокращенные способы сделать это, но работают они одинаково:

```
// Длинный способ
int[] topPlayerScores = new int[] {713, 549, 984};

// Короткий способ
int[] topPlayerScores = { 713, 549, 984 };
```

Инициализация массивов с помощью сокращенного синтаксиса очень распространена, поэтому я буду использовать ее до конца книги. Однако если вы хотите писать более подробно, то не стесняйтесь прибегать к явным формулировкам.

Теперь, когда с синтаксисом объявления разобрались, поговорим о том, как хранятся элементы массива и как к ним обращаться.

Индексация

Элемент массива хранится по порядку, а номер элемента в последовательности называется индексом. Индексация начинается с нуля, а не с единицы. Индекс элемента — это нечто наподобие ссылки на его расположение. В `topPlayerScores` первое целое число `452` расположено по индексу `0`, число `713` — по индексу `1` и `984` — по индексу `2` (рис. 4.15).

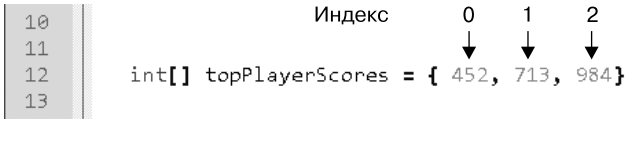


Рис. 4.15

Чтобы обратиться к отдельным значениям, используется оператор индекса, который представляет собой пару квадратных скобок с индексом элемента между ними. Например, чтобы получить и сохранить второй

элемент массива в topPlayerScores, мы должны применить имя массива, за которым следуют скобки и индекс 1:

```
// Установлено значение 713
int score = topPlayerScores[1];
```

Оператор индекса также может использоваться для непосредственного изменения значения массива, как и любая другая переменная, или даже передаваться как выражение:

```
topPlayerScores[1] = 1001;
Debug.Log(topPlayerScores[1]);
```

Теперь в массиве topPlayerScores лежат значения 452, 1001 и 984.

Выход за диапазон

При создании массивов количество элементов фиксируется сразу и не может быть изменено, то есть мы не можем получить доступ к несуществующему элементу. В примере topPlayerScores длина массива равна 3, поэтому диапазон допустимых индексов составляет от 0 до 2. Любой индекс, равный 3 или выше, выходит за пределы диапазона массива и выбрасывает в консоли исключение IndexOutOfRangeException (рис. 4.16).

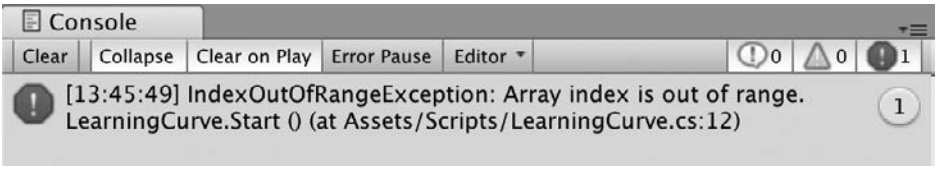


Рис. 4.16

Хорошие практики программирования гласят, что нам следует избегать выходов диапазона, проверяя, находится ли нужное значение в пределах диапазона индекса массива. Об этом поговорим в разделе «Операторы итерации».

Массивы не единственные типы коллекций, имеющиеся в C#. В следующем подразделе мы рассмотрим списки, которые являются более гибкими и чаще используются в программировании.

Списки

Списки тесно связаны с массивами, в них несколько значений одного типа собрано в одной переменной. С ними довольно легко работать в плане добавления, удаления и обновления элементов, но элементы списка хранятся не по порядку. Иногда это приводит к понижению производительности по сравнению с массивами.

Под производительностью понимается мера того, сколько времени и сил компьютера занимает данная операция. В настоящее время компьютеры работают быстро, но все равно бывает, что с трудом справляются с большими играми или приложениями.

Базовый синтаксис

Переменная типа «список» определяется следующим образом:

- ключевое слово `List`, тип элементов, указываемый в треугольных скобках, и уникальное имя;
- ключевое слово `new` для инициализации списка в памяти с ключевым словом `List` и указанием типа элементов между треугольными скобками;
- пара скобок и точка с запятой.

Шаблон объявления выглядит так:

```
List<elementType> name = new List<elementType>();
```

Длину списка всегда можно изменить, поэтому в момент создания нет необходимости указывать, сколько элементов в нем будет.

Как и массивы, списки можно инициализировать сразу при объявлении, перечисляя значения элементов в паре фигурных скобок:

```
List<elementType> name = new List<elementType>() { value1, value2 };
```

Элементы хранятся в том порядке, в котором были добавлены, индексируются с нуля, и к ним можно обращаться с помощью оператора

индекса. Создадим собственный список, чтобы проверить базовую функциональность этого класса.

Время действовать. Составляем список членов группы

В качестве разминки составим список членов группы в вымышленной ролевой игре.

1. Создайте новый список с элементами типа `string` и именем `questPartyMembers` и инициализируйте тремя именами персонажей.
2. Выведите в консоль количество членов группы в списке, используя метод `Count`.
3. Сохраните файл и запустите игру в Unity (рис. 4.17).

```
 7     // Use this for initialization
 8     void Start()
 9     {
10         List<string> questPartyMembers = new List<string>()
11         { "Grim the Barbarian", "Merlin the Wise", "Sterling the Knight"};
12
13         Debug.LogFormat("Party Members: {0}", questPartyMembers.Count);
14     }
15 }
```

Рис. 4.17

Мы создали новый список под названием `questPartyMembers`, который теперь содержит три строковых значения, и использовали метод `Count` из класса `List` для вывода количества элементов (рис. 4.18).

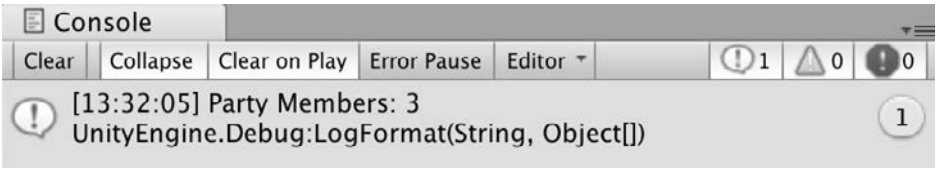


Рис. 4.18

Очень полезно знать, сколько элементов находится в списке, однако в большинстве случаев одной лишь информации недостаточно. Нам нужна возможность изменять наши списки по мере необходимости, что мы и обсудим далее.

Часто используемые методы

К элементам списка можно обращаться и изменять их, как и в массивах, с помощью оператора индекса, если индекс находится в пределах диапазона списка. Однако у класса `List` есть множество методов, предоставляющих дополнительные функции: добавление, вставку и удаление элементов.

Все в том же списке `questPartyMembers` добавим в команду нового участника:

```
questPartyMembers.Add("Craven the Necromancer");
```

Метод `Add` добавляет новый элемент в конец списка, в результате чего количество `questPartyMembers` становится равным четырем, а порядок элементов — следующим:

```
{ "Grim the Barbarian", "Merlin the Wise", "Sterling the Knight",
"Craven the Necromancer"};
```

Чтобы добавить элемент на какое-то определенное место в списке, мы можем передать методу `Insert` индекс и значение, которое хотим добавить:

```
questPartyMembers.Insert(1, "Tanis the Thief");
```

Когда элемент вставляется по указанному индексу, индексы всех далее стоящих элементов в списке увеличиваются на `1`. В нашем примере `Thanis the Thief` теперь имеет индекс `1`, `Merlin the Wise` передвинулся на индекс `2` вместо `1` и т. д.:

```
{ "Grim the Barbarian", "Tanis the Thief", "Merlin the Wise",
"Sterling the Knight", "Craven the Necromancer"};
```

Удалить элемент так же просто — мы передаем методу индекс или буквальное значение, и дальше все происходит само:

```
// Оба метода удаляют элемент
questPartyMembers.RemoveAt(0);
questPartyMembers.Remove("Grim the Barbarian");
```

После всех правок список `questPartyMembers` теперь содержит следующие элементы, проиндексированные от `0` до `3`:

```
{ "Tanis the Thief", "Merlin the Wise", "Sterling the Knight",
"Craven the Necromancer"};
```

У класса List есть еще много методов, которые позволяют проверять значения, находить и сортировать элементы, а также работать с диапазонами. Полный список методов и их описания можно найти здесь: docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.list-1?view=netframework-4.7.2.

Списки отлично подходят для хранения одиночных элементов, но бывают случаи, когда вам нужно хранить информацию или данные, содержащие более одного значения. Здесь на помощь приходят словари.

Словари

Тип `Dictionary` несколько отличается от массивов и списков, поскольку в его элементах хранятся пары значений, а не одиночные значения. Эти элементы называются парами «ключ — значение»: ключ действует как индекс для соответствующего ему значения. В отличие от массивов и списков, словари не упорядочены. Но после того, как они будут созданы, их можно сортировать и упорядочивать различными способами.

Базовый синтаксис

Объявление словаря выполняется почти так же, как объявление списка, но с одной дополнительной деталью: в треугольных скобках надо указать тип и для ключа, и для значения:

```
Dictionary<keyType, valueType> name = new Dictionary<keyType,
valueType>();
```

Чтобы инициализировать словарь парами «ключ — значение», выполните следующие действия.

- Добавьте еще одну пару фигурных скобок в конце объявления.
- Запишите каждый элемент в свою пару фигурных скобок, разделив ключ и значение запятой.
- Разделяйте элементы запятой, кроме последнего элемента, для которого запятая необязательна:

```
Dictionary<keyType, valueType> name = new Dictionary<keyType, valueType>()
{
```

```
    {key1, value1},
    {key2, value2}
};
```

Важное замечание, касающееся выбора значений ключей: каждый ключ должен быть уникальным и их нельзя изменить. Если вам нужно обновить ключ, то измените его значение в объявлении переменной или удалите всю пару «ключ — значение» и добавьте другую в коде.

Как и в случае с массивами и списками, словари можно без проблем инициализировать в одну строку. Однако записывать каждую пару «ключ — значение» в своей строке, как в предыдущем примере, — хорошая привычка, полезная как для удобочитаемости, так и для душевного равновесия.

Время действовать. Настраиваем инвентарь

Создадим словарь для хранения предметов, которые может носить персонаж.

1. Объявите словарь с типом ключа `string` и типом значения `int` и назовите его `itemInventory`.
2. Инициализируйте строкой `new Dictionary <string, int>()` и добавьте три пары «ключ — значение» на свое усмотрение. Убедитесь, что каждый элемент заключен в фигурные скобки.
3. Добавьте вывод в консоль свойства `itemInventory.Count`, чтобы мы могли видеть, как хранятся элементы.
4. Сохраните файл и запустите игру (рис. 4.19).

```
 7     // Use this for initialization
 8     void Start()
 9     {
10         Dictionary<string, int> itemInventory = new Dictionary<string, int>()
11         {
12             { "Potion", 5 },
13             { "Antidote", 7 },
14             { "Aspirin", 1 }
15         };
16
17         Debug.LogFormat("Items: {0}", itemInventory.Count);
18     }
19 }
```

Рис. 4.19

Мы создали новый словарь с именем `itemInventory`, в котором изначально есть три пары «ключ — значение». В качестве ключей у нас строки, а значения — целые числа. Мы вывели на экран количество элементов в словаре `itemInventory` в данный момент (рис. 4.20).

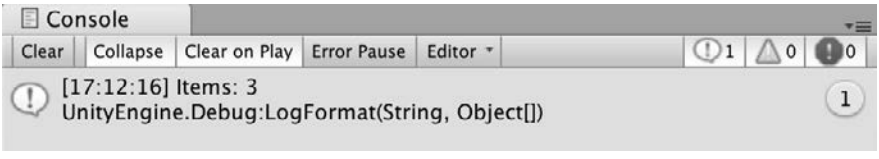


Рис. 4.20

Как и в списках, нам нужна возможность делать нечто большее, чем просто выводить на экран количество пар «ключ — значение» в данном словаре. Добавление, удаление и обновление этих значений мы рассмотрим в следующем разделе.

Работа со словарными парами

Пары «ключ — значение» можно добавлять и удалять; также к ним можно обращаться с помощью методов индекса и класса. Чтобы получить значение элемента, используйте оператор индекса с указанием ключа. В следующем примере ключу `numberOfPotions` будет присвоено значение `5`:

```
int numberOfPotions = itemInventory["Potion"];
```

Значение элемента можно изменить тем же способом. Теперь значение `Potion` будет равно `10`:

```
itemInventory["Potion"] = 10;
```

Добавлять элементы в словарь можно двумя способами: с помощью метода `Add` и оператора индекса. Метод `Add` принимает ключ и значение и создает новый элемент, если переданные методу типы соответствуют структуре:

```
itemInventory.Add("Throwing Knife", 3);
```

Если для присвоения значения ключу используется обращение по индексу, которого нет в словаре, то компилятор автоматически до-

бавит в словарь новую пару «ключ — значение». Например, если мы хотим добавить новый элемент `"Bandage"`, то можно просто написать вот так:

```
itemInventory["Bandage"] = 5;
```

Это важный момент, касающийся ссылок на пары «ключ — значение»: лучше убедиться в наличии элемента, прежде чем пытаться получить к нему доступ, чтобы избежать ошибочного добавления новых пар «ключ — значение». Проще всего использовать метод `ContainsKey` с оператором `if`, поскольку `ContainsKey` возвращает логическое значение в зависимости от того, существует ли в словаре данный ключ. В следующем примере мы проверяем, есть ли ключ `"Aspirin"` в словаре:

```
if(itemInventory.ContainsKey("Aspirin"))
{
    itemInventory["Aspirin"] = 3;
}
```

Наконец, пару «ключ — значение» можно удалить из словаря с помощью метода `Remove`, который принимает в качестве аргумента сам ключ:

```
itemInventory.Remove("Antidote");
```

Как и у списков, у словарей есть множество методов и функций, облегчающих процесс разработки, но мы не можем здесь описать их все. Если вам интересно, то почитайте официальную документацию по адресу docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.dictionary-2?view=netframework-4.7.2.

Итак, теперь в нашем арсенале есть коллекции, поэтому пришло время снова ответить на контрольные вопросы, чтобы убедиться в готовности перейти к следующей серьезной теме: операторам итерации.

Контрольные вопросы 2. Все о коллекциях

1. Что такое элемент массива или списка?
2. Каков порядковый номер первого элемента в массиве или списке?

3. Может ли один массив или список хранить разные типы данных?
4. Как добавить в массив новые элементы, чтобы можно было хранить больше данных?

Поскольку коллекции — это группы или списки элементов, должен быть эффективный способ обрабатывать их. К счастью, в C# есть несколько операторов итерации, о которых мы поговорим в следующем разделе.

Операторы итерации

Доступ к отдельным элементам коллекции мы получали через оператор индекса и методы этого типа коллекции, но что, если нам нужно перебрать всю коллекцию поэлементно? В программировании это называется итерацией, и язык C# предоставляет несколько типов операторов, которые позволяют перебирать элементы коллекции. Операторы итерации похожи на методы тем, что в них содержится блок кода, который нужно выполнить, но, в отличие от методов, эти операторы выполняют свой код многократно.

Цикл for

Цикл `for` чаще всего используется, когда необходимо выполнить некий блок кода определенное количество раз, а затем продолжить программу. Сам оператор принимает три выражения, каждое из которых выполняет определенную функцию. Поскольку циклы `for` отслеживают номер текущей итерации, они лучше всего подходят для массивов и списков.

А вот как выглядит шаблон оператора цикла:

```
for (initializer; condition; iterator)
{
    Выполнить эти строки кода;
}
```

Разберемся, как это работает.

- Ключевое слово `for` начинает оператор, а за ним следует пара круглых скобок. Внутри круглых скобок находятся привратники: инициализатор, условие и итератор.
- Цикл начинается с инициализатора — это локальная переменная, которая отслеживает, сколько раз выполнился цикл. Обычно отсчет начинается с 0, поскольку у коллекций тоже индексация с нуля.
- Затем проверяется выражение условия и, если оно истинно, выполняется итератор. Он используется для увеличения или уменьшения инициализатора, из чего следует, что в следующем проходе цикла значение инициализатора будет уже другим.

На словах это звучит сложно, поэтому рассмотрим практический пример со списком `questPartyMembers`, который мы создали ранее:

```
List<string> questPartyMembers = new List<string>()
{ "Grim the Barbarian", "Merlin the Wise", "Sterling the Knight"};

for (int i = 0; i < questPartyMembers.Count; i++)
{
    Debug.LogFormat("Index: {0} - {1}", i, questPartyMembers[i]);
}
```

Еще раз пройдемся по циклу и посмотрим, как он работает.

- В качестве инициализатора в цикле `for` взята локальная переменная типа `int` с именем `i` с начальным значением `0`.
- Чтобы гарантировать, что мы никогда не выйдем за пределы допустимого диапазона, цикл будет запускать следующую итерацию только в случае, если `i` меньше, чем количество элементов в списке `questPartyMembers`.
- Наконец, значение `i` увеличивается на 1 с помощью оператора `++` каждый раз, когда цикл выполняет один проход.
- Внутри цикла `for` мы выводим индекс и элемент списка по этому индексу, используя `i`. Обратите внимание: `i` буквально проходит по индексам элементов коллекции, поскольку оба начинаются с 0 (рис. 4.21).

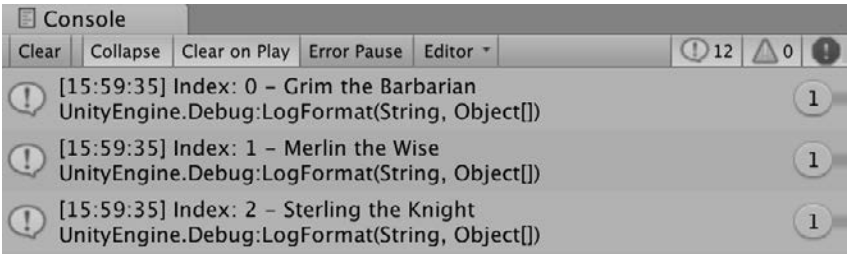


Рис. 4.21

Традиционно буква i обычно используется как имя переменной инициализатора. Если у вас есть вложенные циклы for, то в качестве имен переменных будут использоваться буквы j, k, l и т. д.

Попробуем новый оператор итерации на одной из имеющихся коллекций.

Время действовать. Ищем элемент

Переберем список `questPartyMembers` и посмотрим, сможем ли мы определить, встретится ли определенный элемент. Если встретим его, выведем в консоль сообщение.

1. Добавьте оператор `if` под выводом сообщения в цикле `for`.
2. В условии оператора `if` проверьте, равен ли элемент `questPartyMember` значению `Merlin the Wise`.
3. Если это так, то выведите сообщение на свое усмотрение (рис. 4.22).

Вывод в консоль должен выглядеть почти так же, за исключением того, что появилось одно дополнительное сообщение, которое выводится один раз, когда цикл находит Мерлина. В частности, когда во втором цикле переменная `i` была равна `1`, срабатывал оператор `if` и выводились два сообщения вместо одного (рис. 4.23).

Стандартный цикл `for` в правильной ситуации может быть весьма полезен, но в программировании редко бывает лишь один способ выполнить задачу. Поэтому знакомьтесь: оператор `foreach`.

```
// Start is called before the first frame update
void Start()
{
    List<string> questPartyMembers = new List<string>()
    { "Grim the Barbarian", "Merlin the Wise", "Sterling the Knight" };

    for (int i = 0; i < questPartyMembers.Count; i++)
    {
        Debug.LogFormat("Index: {0} - {1}", i, questPartyMembers[i]);

        if(questPartyMembers[i] == "Merlin the Wise")
        {
            Debug.Log("Glad you're here Merlin!");
        }
    }
}
```

Рис. 4.22

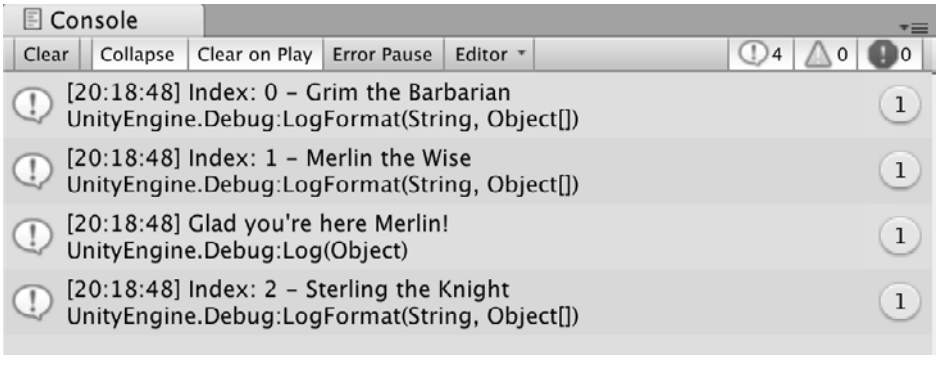


Рис. 4.23

Цикл foreach

Цикл `foreach` берет каждый элемент в коллекции и сохраняет его в локальной переменной для дальнейшей обработки внутри оператора. Для правильной работы тип локальной переменной должен соответствовать типу элемента коллекции. Цикл `foreach` можно использовать с массивами и списками, но особенно полезен он при работе со словарями, поскольку для работы ему не нужен числовой индекс.

Шаблон цикла `foreach` выглядит следующим образом:

```
foreach(elementType localName in collectionVariable)
{
    Выполнить эти строки кода
}
```

Вернемся к примеру с `questPartyMembers` и проведем перекличку всех его элементов:

```
List<string> questPartyMembers = new List<string>()
{ "Grim the Barbarian", "Merlin the Wise", "Sterling the Knight"};

foreach(string partyMember in questPartyMembers)
{
    Debug.LogFormat("{0} - Here!", partyMember);
}
```

Все это работает следующим образом.

- Объявляем тип элемента `string`, что соответствует значениям в списке `questPartyMembers`.
- Создается локальная переменная `partyMember`, в которой хранятся все элементы, перебираемые циклом.
- Наконец, заканчивает синтаксис ключевое слово `in`, за которым следует коллекция, которую мы хотим перебрать, в данном случае `questPartyMembers` (рис. 4.24).

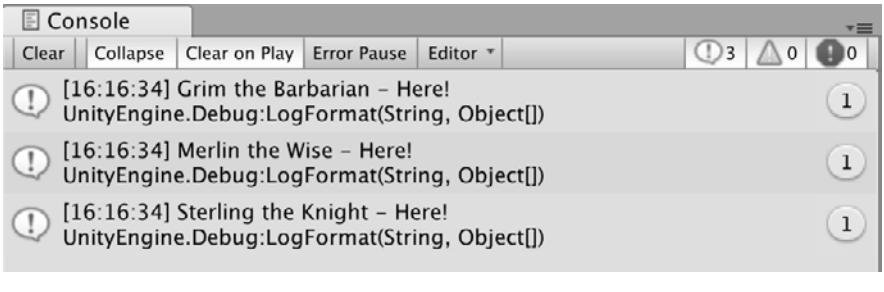


Рис. 4.24

Такой вариант намного проще, чем цикл `for`. Однако необходимо упомянуть несколько важных особенностей работы со словарями, а имен-

но принципы работы с парами «ключ — значение» как с локальными переменными.

Проход по парам «ключ — значение»

Чтобы записать пару «ключ — значение» в локальную переменную, пригодится метко названный тип `KeyValuePair`, который позволяет задать ключи и значения соответствующим типам словаря. Поскольку `KeyValuePair` является типом, в виде локальной переменной он работает так же, как и любой другой тип элемента.

Например, пройдемся по словарю `itemInventory`, который мы создали ранее, и выведем каждую пару «ключ — значение», скажем описание товара в магазине:

```
Dictionary<string, int> itemInventory = new Dictionary<string, int>()
{
    { "Potion", 5},
    { "Antidote", 7},
    { "Aspirin", 1}
};

foreach(KeyValuePair<string, int> kvp in itemInventory)
{
    Debug.LogFormat("Item: {0} - {1}g", kvp.Key, kvp.Value);
}
```

Мы завели локальную переменную типа `KeyValuePair` с именем `kvp`, которое соответствует общепринятому соглашению об именах в программировании, как и имя переменной-счетчика `i` в цикле `for`, и установили типы ключей и значений на `string` и `int`, как в `itemInventory`.

Чтобы получить доступ к ключу и значению локальной переменной kvp, мы используем свойства Key и Value типа KeyValuePair соответственно.

В данном примере ключи являются строками, а значения — целыми числами. Для нас это будет название и стоимость товара (рис. 4.25).

Если вы любите приключения, то попробуйте выполнить следующее дополнительное задание, чтобы закрепить вновь обретенные знания.

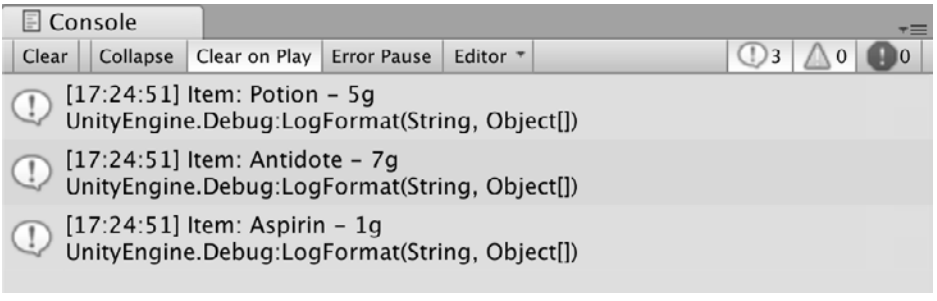


Рис. 4.25

Испытание героя. Ищем подходящие предметы

Создайте переменную, в которой хранится количество золота вашего вымышленного персонажа, и попробуйте добавить оператор `if` внутри цикла `foreach`, чтобы проверить, какие из предметов вы можете себе позволить. Подсказка: используйте атрибут `kvp.Value`, чтобы сравнить цены с содержимым вашего кошелька.

Цикл while

Цикл `while` аналогичен оператору `if` в том смысле, что выполняется до тех пор, пока истинно одно выражение или условие. Сравнение значений и логические переменные могут использоваться в качестве условий `while`, и их можно изменять с помощью оператора `NOT`.

Синтаксис цикла `while` гласит: *«Пока мое условие истинно, пусть этот блок кода выполняется бесконечно»*:

```
initializer
while (condition)
{
    Выполнить эти строки кода;
    итератор;
}
```

В циклах `while` обычно вводят переменную-инициализатор, как и в цикле `for`, и вручную увеличивают или уменьшают ее в конце кода цикла. В этом случае инициализатор чаще всего является частью условия цикла.

Время действовать. Отслеживаем жизни игроков

Рассмотрим стандартный случай, когда нам нужно выполнять код, пока игрок жив, а затем выводить сообщение, если вдруг с ним что-то случилось.

1. Создайте переменную-инициализатор с именем `playerLives`, типа `int`, и задайте ей значение `3`.
2. Объявите цикл `while`, в условии которого проверяется, больше ли `playerLives`, чем `0` (то есть жив ли игрок).
3. Внутри цикла `while` выведите что-нибудь, чтобы мы знали, что персонаж все еще жив, а затем уменьшите `playerLives` на 1 с помощью оператора `--`.

Увеличение и уменьшение значения на 1 называется инкрементом и декрементом соответственно (оператор — уменьшает значение на 1, а ++ увеличивает его на 1).

4. Выведите сообщение после фигурных скобок цикла `while`, чтобы вывести в консоль что-нибудь, когда жизни закончатся (рис. 4.26).

```
 7     // Use this for initialization
 8     void Start()
 9     {
10         int playerLives = 3;
11
12         while(playerLives > 0)
13         {
14             Debug.Log("Still alive!");
15             playerLives--;
16         }
17
18         Debug.Log("Player KO'd...");
19     }
20 }
```

Рис. 4.26

Если `playerLives` начинается с `3`, цикл `while` выполнится три раза. В каждом проходе цикла выводится сообщение `"Still alive!"` и значение переменной `playerLives` уменьшается на 1. Когда цикл `while`

запускается в четвертый раз, условие не выполняется, поскольку `playerLives` к тому моменту равняется `0`, поэтому блок кода пропускается и выводится последнее сообщение (рис. 4.27).

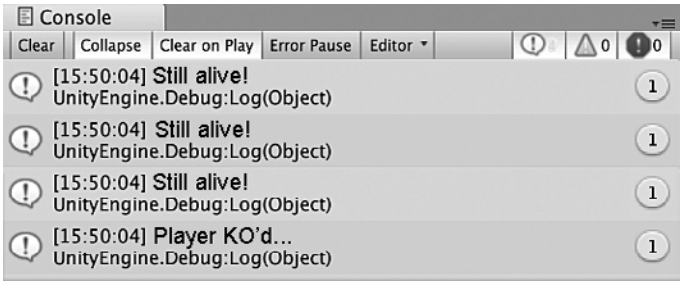


Рис. 4.27

А что произойдет, если цикл никогда не перестанет выполняться? Этот вопрос мы обсудим в следующем разделе.

Бесконечность не предел!

Прежде чем подвести итог главы, нам нужно понять одну чрезвычайно важную концепцию, связанную с операторами итерации: *бесконечные циклы*. Название говорит само за себя: в таком цикле условия не позволяют ему прекратить выполнение и передать управление основной программе.

Бесконечное выполнение обычно возникает в циклах `for` и `while`, когда итератор не увеличивается и не уменьшается. Если бы мы убрали из цикла строку с уменьшением переменной `playerLives`, то Unity завис бы и/или «вылетел», поскольку в этом случае `playerLives` всегда будет равно 3 и цикл будет работать вечно.

Итераторы не единственные виновники проблем. Можно и условие в цикле `for` настроить так, что оно никогда не перестанет срабатывать и снова получится бесконечный цикл. Если бы в примере с членами группы из пункта «Проход по парам “ключ — значение”» мы установили условие цикла `for` равным `i < 0` вместо `i < questPartyMembers.Count`, то `i` всегда была бы больше или равна `0` и Unity бы сперва впал в бесконечный цикл, а затем «вылетел» бы.

Подведем итоги

Заканчивая главу, подумаем о том, чего мы достигли и что теперь можем создать с помощью новых знаний. Мы знаем, как использовать простые проверки `if-else` и более сложные операторы `switch`, позволяющие принимать решения в коде. Мы можем создавать переменные-коллекции значений: массивы, списки или пары «ключ — значение» (словари). Коллекции позволяют эффективно хранить сложные и сгруппированные данные. Мы даже можем подобрать оператор цикла, наиболее удачно подходящий для каждого типа коллекции, тщательно избегая впадений в бесконечный цикл. Если ваш мозг слегка вскипает от избытка информации, то это нормально. Логическое, последовательное мышление — часть обучения программированию.

В следующей главе мы завершим изучение основ программирования на C#, рассмотрев работу с классами, структурами и концепцию **объектно-ориентированного программирования (ООП)**. Мы вложим в эти темы все, что узнали до сих пор, и подготовимся к первому настоящему погружению в управление объектами движка Unity.

5 Работа с классами, структурами и ООП

Разумеется, цель книги заключается не в том, чтобы ваша голова взорвалась от переизбытка информации. Тем не менее темы, которые мы рассмотрим в этой главе, переведут вас из комнаты для новичков в мир **объектно-ориентированного программирования (ООП)**. До этого момента мы полагались исключительно на предопределенные типы переменных, которые являются частью языка C#, а именно встроенные строки, списки и словари. Эти типы являются классами, вследствие чего мы можем создавать их и использовать их свойства через точечную нотацию. Однако у использования встроенных типов есть один вопиющий недостаток — невозможность отклониться от схем, определенных языком C#.

Создание классов дает вам свободу определять и настраивать свои шаблоны для проектирования программы, собирать информацию и управлять действиями, характерными для вашей игры или приложения. По сути, пользовательские классы и ООП — это ключи от царства программирования. Без них уникальных программ было бы мало.

В этой главе мы получим практический опыт создания классов с нуля и поговорим о внутреннем устройстве переменных, конструкторов и методов класса. Кроме того, мы познакомимся с различиями между объектами ссылочного типа и типа значения, а также с тем, чем эти концепции полезны в Unity. По мере продвижения по главе мы подробно рассмотрим следующие темы:

- определение классов;
- объявление структур;
- объявление и использование структур;
- общие сведения о ссылочных типах и типах значения;
- основы ООП;
- применение ООП в Unity.

Определение класса

В главе 2 мы кратко поговорили о том, что классы являются схемами объектов, и я упомянул, что их можно рассматривать как пользовательские типы переменных.

Мы также узнали, что сценарий `LearningCurve` — это класс, но особый, который Unity позволяет прикреплять к объектам на сцене. Главное, что нужно помнить о классах, — это то, что они относятся к *ссылочным типам*, то есть когда они назначаются или передаются другой переменной, создается ссылка на исходный объект, а не на новую копию. К этому мы еще вернемся после обсуждения структур, но прежде вам нужно будет понять основы создания классов.

Базовый синтаксис

На мгновение забудем, как классы и сценарии работают в Unity, и сосредоточимся на том, как создавать и использовать их в C#.

Если вы помните схему, которую мы набросали ранее, то классы создаются с помощью ключевого слова `class`, как показано ниже:

```
accessModifier class UniqueName
{
    Переменные
    Конструкторы
    Методы
}
```

Любые переменные или методы, объявленные внутри класса, принадлежат ему, и обращаться к ним можно через уникальное имя данного класса.

Чтобы все примеры в этой главе хорошо вплетались в канву, заданную нами ранее, мы создадим и будем далее развивать простой класс `Character`, который часто встречается в играх. Вдобавок я перестану приводить скриншоты с кодом, чтобы вы привыкли читать и интерпретировать код, так сказать, «вживую».

Но первое, что нам понадобится, — собственный класс, поэтому создадим его.

Время действовать. Создаем класс персонажа

Нам понадобится класс, на котором мы будем практиковаться, чтобы понять, как все устроено. А раз так, то создадим новый сценарий C# и начнем с нуля.

1. Щелкните правой кнопкой мыши на папке Scripts, выберите команду Create, а затем C# Script.
2. Назовите новый сценарий `Character`, откройте его в Visual Studio и удалите весь автоматически сгенерированный код после строки `using UnityEngine`.
3. Объявите `public class` с именем `Character`, после имени добавьте пару фигурных скобок, а затем сохраните файл. Код вашего класса должен выглядеть следующим образом:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Character
{

}
```

Теперь `Character` — шаблон публичного класса. Это значит, что любой класс в проекте может обращаться к нему для создания персонажей. Однако это всего лишь своего рода инструкция. А для создания персонажа потребуется еще один шаг. Данный этап называется *созданием экземпляра*, и следующий подраздел посвящен именно ему.

Создание экземпляра объекта класса

Создание экземпляра — это создание нового объекта из заранее известного набора инструкций. Новый объект называется экземпляром. Если классы — это чертежи, то экземпляры — это дома, построенные по данному чертежу. Каждый новый экземпляр `Character` — отдельный объект класса, точно так же, как два дома, построенные по одним и тем же инструкциям, являются двумя разными домами. Происходящее с одним из них не влияет на другой.

В предыдущей главе мы создали списки и словари, которые являются классами, с помощью их типов и ключевого слова `new`. То же самое можно делать и с пользовательскими классами, такими как `Character`, что мы и рассмотрим дальше.

Время действовать. Создаем новый персонаж

Мы объявили класс `Character` как публичный, а это значит, что экземпляр `Character` может быть создан в любом другом классе. Поскольку у нас уже есть сценарий `LearningCurve`, объявим новый персонаж в методе `Start()`.

Объявите новую переменную типа `Character` в методе `Start()` сценария `LearningCurve`:

```
Character hero = new Character();
```

Пошагово посмотрим, что здесь к чему.

- Мы задали тип переменной `Character`; это значит, что переменная является экземпляром данного класса.
- Имя переменной — `hero`, и она создается с использованием ключевого слова `new`, за которым следуют имя класса `Character` и две круглые скобки. Здесь в памяти программы создается фактический экземпляр, даже если класс сейчас пуст.

Мы можем использовать переменную `hero`, как и любой другой объект, с которым работали до сих пор. Когда у класса `Character` будут собственные переменные и методы, мы сможем получить к ним доступ из переменной `hero` с помощью точечной нотации.

Для создания нового персонажа можно также использовать предполагаемое объявление:

```
var hero = new Character();
```

Однако наш класс персонажа не очень-то полезен без полей класса, с которыми можно было бы работать. В следующих нескольких подразделах мы добавим поля классов и многое другое.

Добавление полей класса

Добавление переменных или полей в пользовательский класс ничем не отличается от того, что мы уже делали в сценарии `LearningCurve`. Мы задействуем уже известные концепции, включая модификаторы доступа, область действия переменных и присвоение значений. Однако любые переменные, принадлежащие классу, создаются с экземпляром класса, а это значит, что если им не присвоены значения, то они будут по умолчанию равны нулю (zero), или `null`. В целом выбор начальных значений сводится к тому, какая информация будет храниться в переменной:

- если некая переменная должна иметь одно и то же начальное значение при создании экземпляра класса, то можно задать начальное значение сразу;
- если переменную необходимо настраивать по-своему в каждом экземпляре класса, то оставьте ее значение неназначенным и используйте конструктор класса (о них мы поговорим позже).

Каждому классу персонажа потребуется несколько основных полей. Далее нам предстоит добавить их.

Время действовать. Конкретизируем детали персонажа

Добавим две переменные, в которых будем хранить имя персонажа и изначальное количество очков опыта.

1. Добавим две публичные переменные внутри фигурных скобок класса `Character` — строковую переменную для имени и целочисленную переменную для очков опыта.
2. Оставьте значение `name` пустым, а очкам опыта задайте значение `0`, чтобы каждый персонаж стартовал с начала:

```
public class Character
{
    public string name;
    public int exp = 0;
}
```

3. Добавьте в сценарий `LearningCurve` вывод в консоль сразу после инициализации экземпляра `Character`. Выведите переменные `name` и `exp` нового персонажа, используя точечную нотацию:

```
Character hero = new Character();
Debug.LogFormat("Hero: {0} - {1} EXP", hero.name, hero.exp);
```

Когда инициализируется объект `hero`, его параметру `name` присваивается нулевое значение, которое в консоли отображается как пустое место, а `exp` равно `0`. Обратите внимание: нам не нужно было прикреплять сценарий `Character` к каким-либо объектам `GameObject` в сцене. Мы просто сослались на него в `LearningCurve`, а все остальное сделал Unity. Теперь в консоли будет выводиться информация о нашем герое игры следующим образом (рис. 5.1).

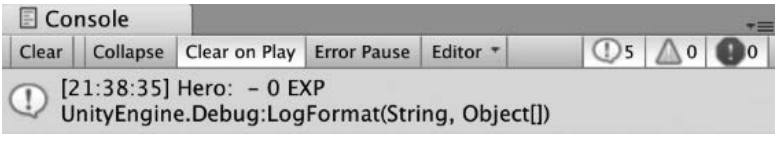


Рис. 5.1

На данный момент наш класс работает, но с пустыми значениями пользы от него пока немного. Попробуем исправить это с помощью так называемого конструктора класса.

Использование конструкторов

Конструкторы классов — это специальные методы, которые запускаются автоматически при создании экземпляра класса, что аналогично запуску метода `Start` в сценарии `LearningCurve`. Конструкторы строят класс в соответствии с его шаблоном:

- если конструктор не указан, то C# создает конструктор по умолчанию. Он устанавливает для всех переменных значения по умолчанию, соответствующие их типам: числовые значения равны нулю, логические значения — `false`, а ссылочные типы (классы) — `null`;

- пользовательские конструкторы можно определять с помощью параметров, как и любой другой метод, и они призваны задавать значения переменных класса при инициализации;
- у класса может быть несколько конструкторов.

Конструкторы — обычные методы, но с некоторыми отличиями. Например, они должны быть публичными, не иметь возвращаемого типа, а имя метода всегда является именем класса. В качестве примера добавим базовый конструктор без параметров к классу `Character` и установим в поле `name` значение, отличное от `null`.

Добавьте этот новый код непосредственно под переменные класса, как показано ниже:

```
public string name;
public int exp = 0;

public Character()
{
    name = "Not assigned";
}
```

Запустите проект в Unity, и вы увидите экземпляр `hero`, созданный через новый конструктор. Параметр `name` будет иметь значение `Not assigned` вместо `null` (рис. 5.2).

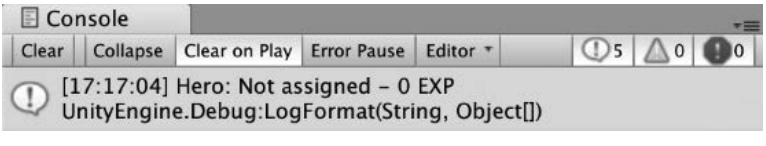


Рис. 5.2

Уже лучше, но нам требуется бóльшая гибкость конструктора класса. Это значит, нам нужна возможность передавать ему значения, чтобы их можно было использовать в качестве начальных значений для экземпляра класса. Так и поступим.

Время действовать. Определяем начальные свойства

Теперь поведение класса `Character` больше похоже на поведение настоящего объекта, но мы можем еще улучшить его, добавив второй

конструктор, который принимает имя при инициализации и устанавливает его в поле имени.

1. Добавьте в `Character` еще один конструктор, который принимает параметр типа `string` с именем `name`.
2. Назначьте параметр переменной `name` класса с помощью ключевого слова `this`. Это называется *перегрузкой конструктора*:

```
public Character(string name)
{
    this.name = name;
}
```

Для удобства у конструкторов часто бывают параметры, имена которых совпадают с именем переменной класса. В этих случаях поможет ключевое слово this, указывающее, какая переменная принадлежит классу. В приведенном здесь примере this.name указывает на переменную name данного класса, а name — это параметр. Если не использовать ключевое слово `this`, то компилятор выдаст предупреждение, поскольку для него эти имена одинаковы.

3. Создайте в сценарии `LearningCurve` новый экземпляр класса `Character` с именем `heroine`. Примените пользовательский конструктор, чтобы передать имя при его инициализации и вывести его в консоль:

```
Character heroine = new Character("Agatha");
Debug.LogFormat("Hero: {0} - {1} EXP", heroine.name, heroine.exp);
```

Когда у класса есть несколько конструкторов или у метода несколько вариантов, Visual Studio покажет набор стрелок во всплывающем окне автозаполнения, которое можно прокручивать с помощью кнопок со стрелками (рис. 5.3).

Рис. 5.3

Теперь мы можем выбирать при инициализации нового класса `Character` между базовым и расширенным конструктором. Сам класс `Character` теперь гораздо более гибок и позволяет настраивать разные экземпляры для разных ситуаций (рис. 5.4).

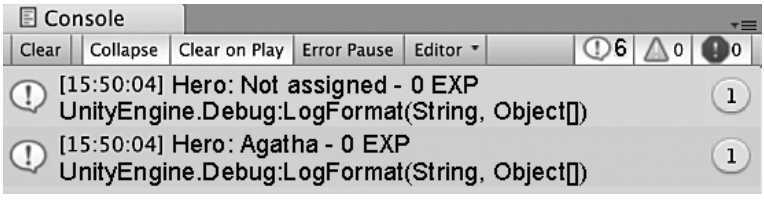


Рис. 5.4

Теперь начинается настоящая работа. Нашему классу нужны методы, чтобы они могли делать нечто полезное, кроме хранения переменных. Ваша следующая задача — применить это на практике.

Объявление методов класса

Добавление методов в пользовательские классы ничем не отличается от добавления их в `LearningCurve`. Однако это прекрасная возможность поговорить о главном правиле хорошего программирования — Don't Repeat Yourself (DRY), эталонном принципе качественного кода. По сути, если вы вдруг осознали, что пишете одну и ту же строку или строки снова и снова, то пришло время переосмыслить и реорганизовать ваш код. Обычно это приводит к созданию нового метода, в котором хранится повторяющийся код, что упрощает его изменение и дальнейшее использование в другом месте.

В программировании это называется абстрагированием метода или функции.

У нас уже есть изрядное количество повторяющегося кода, поэтому посмотрим, получится ли нам повысить удобочитаемость и эффективность наших сценариев.

Время действовать. Выводим данные персонажа

Наши сообщения в консоли — прекрасная возможность абстрагироваться от кода непосредственно в классе `Character`.

1. Добавьте в класс `Character` новый общедоступный метод с возвращаемым типом `void` с названием `PrintStatsInfo`.
2. Скопируйте и вставьте вывод в консоль из класса `LearningCurve` в тело метода.
3. Измените переменные на `name` и `exp`, поскольку теперь на них можно ссылаться напрямую из класса:

```
public void PrintStatsInfo()
{
    Debug.LogFormat("Hero: {0} - {1} EXP", name, exp);
}
```

4. Замените вывод в консоль персонажа, который мы добавили ранее в метод `LearningCurve`, поместив на его место вызов метода `PrintStatsInfo`, и нажмите кнопку Play:

```
Character hero = new Character();
hero.PrintStatsInfo();

Character heroine = new Character("Agatha");
heroine.PrintStatsInfo();
```

Теперь, когда у класса `Character` есть метод, любой экземпляр может свободно обращаться к нему через точечную нотацию. Поскольку `hero` и `heroine` являются отдельными объектами, `PrintStatsInfo` выводит их соответствующие значения `name` и `exp` в консоли.

Это лучше, чем писать вывод в консоль напрямую в LearningCurve. Всегда полезно группировать функциональные возможности в класс и управлять действиями с помощью методов. Это делает код более удобочитаемым, поскольку наши объекты Character отдают команду при выводе сообщений в консоль вместо повторения кода.

Весь класс `Character` приведен на рис. 5.5.

Мы обсудили классы и теперь возьмемся за их более легкий объект-собрат — **структуры**!

```
5   public class Character
6   {
7       public string name;
8       public int exp = 0;
9
10      public Character()
11      {
12          name = "Not assigned";
13      }
14
15      public Character(string name)
16      {
17          this.name = name;
18      }
19
20      public virtual void PrintStatsInfo()
21      {
22          Debug.LogFormat("Hero: {0} - {1} EXP", name, exp);
23      }
```

Рис. 5.5

Объявление структур

Структуры похожи на классы в том смысле, что тоже являются шаблонами объектов, которые мы создаем в программе. Основное отличие состоит в том, что структуры относятся к *типам «значение»*, то есть передаются по значению, а не по ссылке, как классы. Поговорим об этом более подробно ниже. В первую очередь нам необходимо понять, как работают структуры и по каким правилам создаются.

Базовый синтаксис

Структуры объявляются так же, как классы, и могут содержать поля, методы и конструкторы:

```
accessModifier struct UniqueName
{
    Переменные
    Конструкторы
    Методы
}
```

Как и классы, любые переменные и методы принадлежат исключительно структуре, и обращаться к ним можно по ее уникальному имени.

Однако у структур есть несколько ограничений:

- переменные нельзя инициализировать сразу при объявлении структуры, если они не отмечены модификатором `static` или `const`, — об этом подробнее в главе 10;
- конструкторы без параметров не допускаются;
- у структур есть конструктор по умолчанию, который автоматически устанавливает для всех переменных значения по умолчанию в соответствии с их типом.

Каждому персонажу нужно хорошее оружие, и информацию о нем удобнее хранить в структуре, а не в классе. Мы обсудим, почему это так, чуть позже. Но сначала создадим структуру, чтобы поэкспериментировать с ней.

Время действовать. Создаем структуру оружия

Чтобы выполнять квесты, персонажам будет нужно хорошее оружие. Его и возьмем для создания простой структуры.

1. Создайте `public struct` под названием `Weapon` в сценарии `Character`. Обязательно сделайте это *вне* фигурных скобок класса `Character`:
   - добавьте поле `name` типа `string`;
   - теперь добавьте поле `damage` типа `int`.

Можно вкладывать классы и структуры друг в друга, но такой метод обычно не одобряется, поскольку загромождает код.

```
public struct Weapon
{
    public string name;
    public int damage;
}
```

2. Объявите конструктор с параметрами `name` и `damage` и установите поля структуры с помощью ключевого слова `this`:

```
public Weapon(string name, int damage)
{
    this.name = name;
```

```
        this.damage = damage;
    }
```

3. Добавьте вывод в консоль под конструктором, чтобы вывести информацию об оружии:

```
public void PrintWeaponStats()
{
    Debug.LogFormat("Weapon: {0} - {1} DMB", name, damage);
}
```

4. В сценарии `LearningCurve` создайте новую структуру `Weapon`, применяя пользовательский конструктор и ключевое слово `new`:

```
Weapon huntingBow = new Weapon("Hunting Bow", 105);
```

Несмотря на то что структура `Weapon` создана в сценарии `Character`, она находится за пределами фактического объявления класса (фигурные скобки) и поэтому не является частью самого класса. Наш новый объект `huntingBow` работает с пользовательским конструктором и при инициализации задает значения для обоих полей.

Рекомендуется создавать в сценарии не более одного класса. Довольно часто встречаются структуры, которые используются исключительно классом, включенным в файл, как в нашем примере со сценарием Character и структурой Weapon.

Теперь, когда у нас есть пример как ссылочных объектов (класс), так и объектов-значений (структура), пора познакомиться с ними поближе. В частности, необходимо понять, как каждый из этих объектов передается и хранится в памяти.

Общие сведения о ссылочных типах и типах значений

За исключением обозначения при объявлении и начальных значений полей, большой разницы между классами и структурами мы пока не видели. Классы лучше всего подходят для группирования сложных

действий и данных, которые в программе будут меняться. А структуры более уместны для простых объектов и данных, в целом не меняющихся. Помимо сферы использования, они фундаментально различаются в важном моменте, а именно в том, как передаются или присваиваются переменным. Классы являются *ссылочными типами*, а значит, передаются по ссылке. Структуры относятся к *типам значений*, то есть передаются по значению.

Ссылочные типы

Когда инициализируются экземпляры нашего класса `Character`, переменные `hero` и `heroine` не хранят информацию о своем классе, но содержат ссылку на то, где находится объект в памяти программы. Если мы присвоим `hero` или `heroine` другой переменной, то создастся новая ссылка на тот же раздел в памяти, а не новые данные. Такой метод имеет несколько последствий, наиболее важным из которых является то, что если несколько переменных хранят одну и ту же ссылку на память, то изменение одной из них повлияет на все.

В данном случае проще показать на примере, чем объяснить, так что этим и займемся.

Время действовать. Создаем нового героя

Пришло время проверить, относится ли класс `Character` к ссылочному типу.

1. Объявите в сценарии `LearningCurve` новую переменную типа `Character` с именем `hero2`. Присвойте `hero2` значение `hero` и используйте метод `PrintStatsInfo` для вывода их информации.
2. Нажмите кнопку Play и посмотрите на сообщения в консоли:

```
Character hero = new Character();
Character hero2 = hero;

hero.PrintStatsInfo();
hero2.PrintStatsInfo();
```

3. Два журнала отладки будут идентичны, поскольку `hero2` было присвоено значение `hero` при создании. На данном этапе и `hero2`, и `hero` указывают на то, где находится герой в памяти (рис. 5.6).

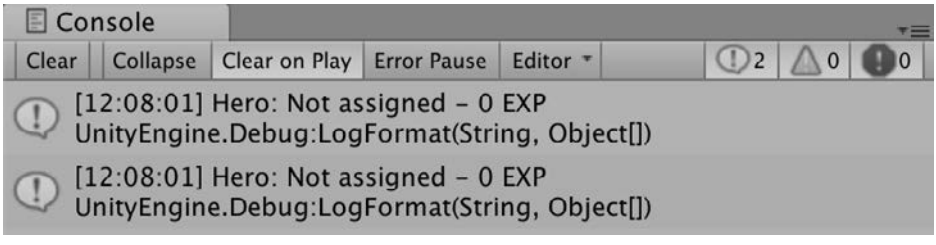


Рис. 5.6

4. Теперь измените имя `hero2` на какое-нибудь другое и снова нажмите кнопку Play:

```
Character hero2 = hero;
hero2.name = "Sir Krane the Brave";
```

Вы увидите, что и у `hero`, и у `hero2` теперь одно и то же имя, даже несмотря на то, что мы меняли только одно из них. Суть в том, что со ссылочными типами нужно обращаться осторожно и они не копируются при назначении новым переменным. Любое изменение одной ссылки влияет на все другие переменные, содержащие ту же ссылку (рис. 5.7).

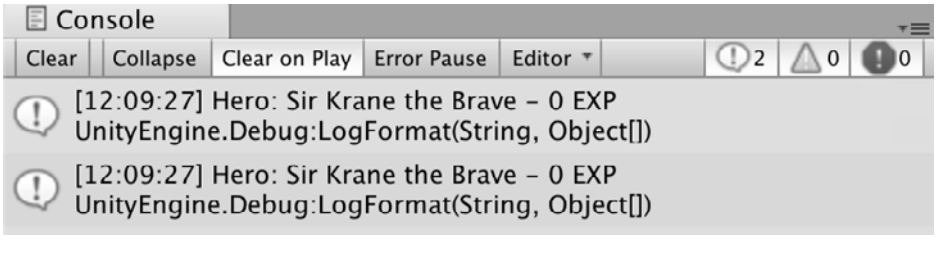


Рис. 5.7

Если вам нужно скопировать класс, то следует либо создать новый отдельный экземпляр, либо подумать об использовании альтернативного типа — структуры. Подробнее о типах-значениях поговорим в следующем подразделе.

Типы-значения

Когда создается объект структуры, все его данные сохраняются в соответствующей переменной без ссылок и каких-либо связей с его ячейкой памяти. Это делает структуры полезными для создания объектов, которые требуется быстро и эффективно скопировать, сохраняя их независимость.

Попробуем проделать это с нашей структурой `Weapon`.

Время действовать. Копируем оружие

Создадим новый объект оружия, скопировав `HuntingBow` в новую переменную и обновив ее данные, чтобы увидеть, влияют ли изменения на обе структуры.

1. Объявите новую структуру `Weapon` в сценарии `LearningCurve` и назначьте ей `huntingBow` в качестве начального значения:

```
Weapon huntingBow = new Weapon("Hunting Bow", 105);
Weapon warBow = huntingBow;
```

2. Выведите данные каждого оружия в консоль:

```
huntingBow.PrintWeaponStats();
warBow.PrintWeaponStats();
```

3. Теперь, когда все настроено, и у `huntingBow`, и у `warBow` будут одинаковые сообщения в консоли, точно так же, как у наших двух персонажей до изменения данных (рис. 5.8).

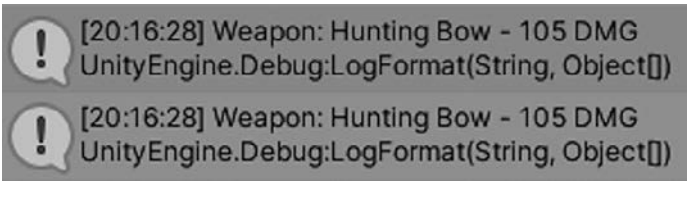


Рис. 5.8

4. Измените поля `warBow.name` и `warBow.damage` на любые другие значения на свое усмотрение и снова нажмите Play:

```
Weapon warBow = huntingBow;

warBow.name = "War Bow";
warBow.damage = 155;
```

В консоли будет видно, что изменились только данные объекта `warBow`, а у `huntingBow` все осталось по-прежнему (рис. 5.9). Вывод из этого примера таков: структуры легко копируются и изменяются как отдельные объекты, в отличие от классов, которые хранят ссылки на исходный объект. Теперь, немного больше понимая, как работают структуры и классы, мы можем начать говорить о парадигме ООП и о том, как она вписывается в среду программирования.

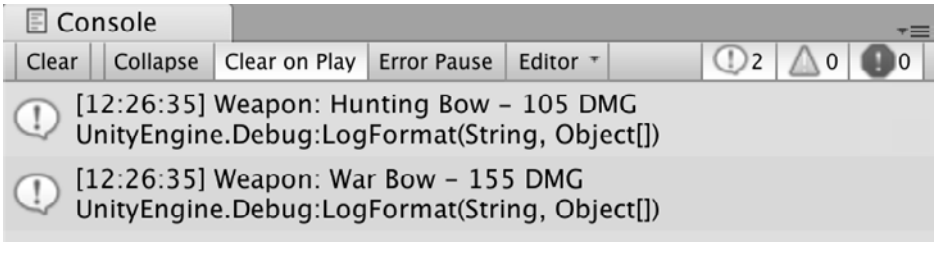


Рис. 5.9

Узнав и на реальных примерах убедившись, как ведут себя ссылочные типы и типы-значения, мы готовы погрузиться в одну из самых важных тем программирования: ООП. Это основная парадигма или архитектура программирования, которую вы будете использовать при написании кода на C#.

Подключаем объектно-ориентированное мышление

Экземпляры классов и структур являются шаблонами для программ, а ООП — архитектура, которая объединяет все это. Говоря об ООП как о парадигме программирования, мы имеем в виду определенные принципы работы и взаимодействия всех компонентов программы. По сути,

ООП фокусируется на объектах, а не на чистой последовательной логике. В ООП главное — это данные, которые хранятся в объектах, управление действиями и, что наиболее важно, общение объектов друг с другом.

В физическом мире все работает аналогичным образом: желая купить что-нибудь освежающее, вы берете банку газировки, а не саму жидкость. Банка — это объект, объединяющий данные и связанные с ними действия в автономный пакет. Однако как в программировании, так и в продуктовом магазине есть определенные правила работы с объектами. Например, есть правила относительно того, кто может получать доступ к объектам. Различные комбинации подобных условий и обобщенных действий приводят к наличию множества объектов вокруг нас. С точки зрения программирования основными столпами ООП являются следующие правила: *инкапсуляция*, *наследование* и *полиморфизм*.

Инкапсуляция

Одно из лучших преимуществ ООП заключается в поддержке инкапсуляции — определении того, насколько переменные и методы объекта доступны для внешнего кода (который иногда называют *вызывающим*). Возьмем, к примеру, нашу банку с газировкой — в торговом автомате возможности взаимодействия с ней ограничены. Поскольку автомат заблокирован, не каждый может просто подойти и взять газировку. Но если вы выполните определенное действие, то вам будет разрешен временный доступ к газировке, но в ограниченном количестве. Если автомат заперт в комнате, то лишь тот, у кого есть ключ от двери, сможет достать банку.

У вас должен возникнуть вопрос: а как установить эти ограничения? Простой ответ заключается в том, что мы уже использовали инкапсуляцию, устанавливая модификаторы доступа для переменных и методов нашего объекта. Если вам нужно освежить память, то просмотрите раздел «Использование модификаторов доступа» в главе 3.

Ниже рассмотрим простой пример инкапсуляции, чтобы понять, как все работает на практике.

Время действовать. Добавляем сброс

Наш класс `Character` является публичным, как и его поля и методы. А что, если нам нужен метод, который бы сбросил данные персонажа до исходных значений? Это может пригодиться, но может и привести к катастрофе, если такой метод будет вызван случайно. Поэтому его лучше сделать приватным.

1. Создайте частный метод с именем `Reset` без возвращаемого значения внутри класса `Character`:
   - задайте для переменных `name` и `exp` значения `Not assigned` и `0` соответственно:

```
private void Reset()
{
    this.name = "Not assigned";
    this.exp = 0;
}
```

2. Попробуйте вызвать метод `Reset` из `LearningCurve` после вывода данных `hero2` (рис. 5.10).

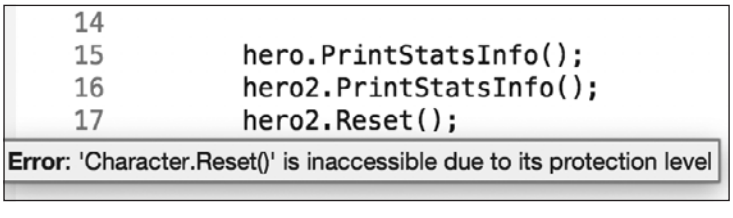


Рис. 5.10

Если вам интересно, не сломалось ли что-нибудь в Visual Studio, то нет. Если мы делаем метод или переменную приватной, то они становятся недоступными для точечной нотации. Если вы введете имя метода вручную и наведете курсор на название `Reset()`, то увидите сообщение об ошибке.

Инкапсуляция допускает более сложные настройки доступности с объектами, но мы пока поприменяем варианты public и private. По мере того как в следующей главе мы начнем конкретизировать наш прототип игры, при необходимости будем добавлять различные модификаторы.

Теперь поговорим о наследовании, которое станет вашим лучшим другом при создании иерархий классов в будущих играх.

Наследование

Как и в жизни, класс C# можно создать по образу и подобию другого класса, передав ему те же атрибуты и методы, а также наделив и своими, уникальными данными. В ООП это называется *наследованием*, и это очень эффективный способ создания связанных классов, не требующий написания одного и того же кода. Снова возьмем пример с газировкой — на рынке есть как обычные газированные напитки, имеющие схожие основные свойства, так и особенные газированные напитки. Особенные газированные напитки обладают одинаковыми основными свойствами, но отличаются брендом или упаковкой. С виду это просто две банки с газировкой, но и их различия очевидны.

Исходный класс обычно называется базовым или родительским, а наследующий класс — производным или дочерним. Любые члены базового класса, отмеченные модификаторами доступа `public`, `protected` или `internal`, автоматически становятся частью производного класса, за исключением конструкторов. Конструкторы классов всегда принадлежат к содержащему их классу, но их можно использовать из производных классов, чтобы свести к минимуму повторяющийся код.

В большинстве игр есть несколько типов персонажей, поэтому создадим новый класс под названием `Paladin`, который наследуется от класса `Character`. Вы можете добавить этот новый класс в сценарий `Character` или создать новый:

```
public class Paladin: Character
{

}
```

Подобно тому как `LearningCurve` наследуется от `MonoBehavior`, мы можем добавить двоеточие и базовый класс, от которого хотим унаследоваться, а C# сделает все остальное. Теперь любые экземпляры `Paladin` будут иметь доступ к атрибутам `name` и `exp` и методу `PrintStatsInfo`.

Обычно рекомендуется создавать новый сценарий для разных классов, а не добавлять их к существующим сценариям. Таким образом вы сможете разделить сценарии и избежать слишком большого количества строк кода в одном файле.

Это прекрасно, но как унаследованные классы создаются конструктором? Об этом подробнее ниже.

Базовые конструкторы

Когда один класс наследуется от другого, они образуют своего рода пирамиду с переменными-членами, которые переходят от родительского класса к любому из его производных потомков. Родительский класс не знает ни одного из своих дочерних элементов, но все потомки знают своего родителя. Однако конструкторы родительского класса можно вызывать непосредственно из дочерних конструкторов с помощью простого синтаксиса:

```
public class ChildClass: ParentClass
{
    public ChildClass(): base()
    {

    }
}
```

Ключевое слово `base` заменяет родительский конструктор: в данном случае конструктор по умолчанию. Однако ввиду того, что `base` заменяет конструктор, а конструктор является методом, дочерний класс может передавать параметры вверх по пирамиде своему родительскому конструктору.

Время действовать. Вызываем базовый конструктор

Поскольку мы хотим, чтобы все объекты класса `Paladin` имели имя, а у класса `Character` уже есть конструктор, в котором это реализовано, мы можем вызвать базовый конструктор непосредственно из класса `Paladin` и избавить себя от необходимости переписывать конструктор.

1. Добавьте в класс `Paladin` конструктор, который принимает строковый параметр `name`:

- используйте двоеточие и ключевое слово `base` для вызова родительского конструктора, передав значение `name`:

```
public class Paladin: Character
{
    public Paladin(string name): base(name)
    {

    }
}
```

2. Создайте новый экземпляр `Paladin` с именем `knight` в сценарии `LearningCurve`:
   - используйте базовый конструктор, чтобы присвоить значение;
   - вызовите метод `PrintStatsInfo` из объекта `knight` и посмотрите на консоль:

```
Paladin knight = new Paladin("Sir Arthur");
knight.PrintStatsInfo();
```

Вывод в консоли будет таким же, как и у других экземпляров `Character`, но с именем, которое мы присвоили конструктору `Paladin` (рис. 5.11). Срабатывая, конструктор `Paladin` передает параметр `name` конструктору `Character`, который устанавливает значение атрибута `name`. По сути, мы использовали конструктор `Character` для инициализации класса `Paladin`, а конструктору `Paladin` оставили только инициализацию уникальных свойств, которых у него на данный момент нет.

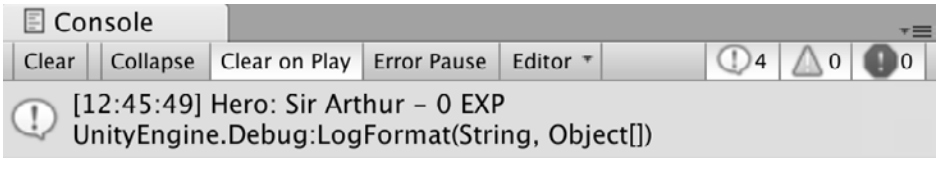


Рис. 5.11

Помимо наследования, будут и другие моменты, когда удобно создать новые объекты путем комбинирования других, уже существующих объектов. Подумайте о конструкторе Lego. Вы ведь не с нуля все делаете,

у вас уже есть блоки разных цветов и структур, с которыми можно работать. В терминах программирования это называется *композицией*, которую мы обсудим в следующем подразделе.

Композиция

Помимо наследования, классы могут состоять из других классов. Возьмем, к примеру, нашу структуру `Weapon`. Класс `Paladin` может содержать внутри себя переменную `Weapon` и иметь доступ ко всем ее свойствам и методам. Сделаем это, выдав нашему `Paladin` стартовое оружие и присвоив его значение в конструкторе:

```
public class Paladin: Character
{
    public Weapon weapon;

    public Paladin(string name, Weapon weapon): base(name)
    {
        this.weapon = weapon;
    }
}
```

Поскольку атрибут `weapon` уникален для `Paladin`, а у `Character` его нет, нам нужно установить его начальное значение в конструкторе. Нам также необходимо обновить экземпляр `knight`, добавив в него переменную `Weapon`. Применим `huntingBow`:

```
Paladin knight = new Paladin("Sir Arthur", huntingBow);
```

Если вы запустите игру сейчас, то не увидите ничего нового, поскольку мы используем метод `PrintStatsInfo` из класса `Character`, который не знает о свойстве `weapon` класса `Paladin`. Чтобы решить эту проблему, придется поговорить о полиморфизме.

Полиморфизм

Полиморфизм — это греческое слово, обозначающее *«множество форм»*, и в ООП данное понятие применяется двумя разными способами.

- Объекты дочернего класса обрабатываются так же, как объекты родительского. Например, массив объектов `Character` может хранить и объекты `Paladin`, поскольку они происходят от `Character`.
- Родительские классы могут помечать методы как `virtual`, это значит, их инструкции могут быть изменены производными классами с помощью ключевого слова `override`. В случае с `Character` и `Paladin` было бы полезно, если бы мы могли в методе `PrintStatsInfo` выводить разные сообщения для каждого из этих классов.

Полиморфизм позволяет дочерним классам сохранять структуру своего родительского класса, но в то же время подстраивать действия под свои конкретные потребности. Возьмем эти новые знания и применим их к выводу информации о персонаже в консоль.

Время действовать. Пробуем функциональные вариации

Подредактируем классы `Character` и `Paladin`, чтобы они выводили в методе `PrintStatsInfo` разные сообщения.

1. Измените метод `PrintStatsInfo` в классе `Character`, добавив ключевое слово `virtual` между `public` и `void`:

```
public virtual void PrintStatsInfo()
{
    Debug.LogFormat("Hero: {0} - {1} EXP", name, exp);
}
```

2. Объявите метод `PrintStatsInfo` в классе `Paladin`, используя ключевое слово `override`:
   - добавьте вывод в консоль свойств `Paladin` любым способом, который вам нравится:

```
public override void PrintStatsInfo()
{
    Debug.LogFormat("Hail {0} - take up your {1}!", name,
    weapon.name);
}
```

Кажется, будто мы создали повторяющийся код, что, как я уже сказал, является плохим тоном, но здесь особый случай. Отметив метод

`PrintStatsInfo` как `virtual` в классе `Character`, мы сообщили компилятору, что этот метод может иметь множество форм в соответствии с вызывающим классом. Объявив переопределенную версию `PrintStatsInfo` в классе `Paladin`, мы добавили поведение, которое применяется только к этому классу. Благодаря полиморфизму нам не нужно выбирать, какую версию `PrintStatsInfo` мы хотим вызвать из объекта `Character` или `Paladin` — компилятор уже знает (рис. 5.12).

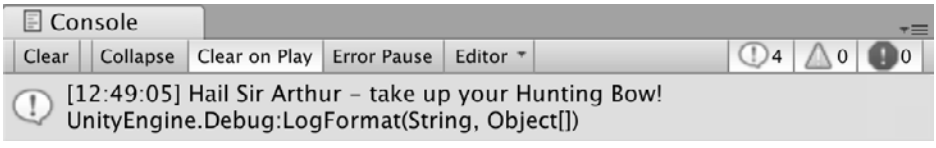


Рис. 5.12

Мы проделали большую работу, поэтому не забудьте прочитать итоговый обзор всего пройденного, прежде чем продолжить!

Итого про ООП

Я знаю, что это было очень сложно. Итак, рассмотрим некоторые из основных моментов ООП и выйдем на финишную прямую:

- ООП — парадигма, предназначенная для группировки связанных данных и действий в объекты. Эти объекты могут взаимодействовать и функционировать независимо друг от друга;
- доступ к членам класса можно настраивать с помощью модификаторов доступа, как и у переменных;
- классы могут наследовать от других классов, создавая нисходящие иерархии родительско-дочерних отношений;
- классы могут быть членами других типов классов или структур;
- классы могут переопределять любые родительские методы, помеченные маркером `virtual`, позволяя им выполнять настраиваемые действия, сохраняя при этом тот же шаблон.

ООП не единственная парадигма программирования, которую можно использовать в C#. В этой статье можно найти практические объяснения других основных подходов: cs.lmu.edu/~ray/notes/paradigms.

Все об ООП, что вы изучили в данной главе, напрямую применимо к миру C#. Тем не менее нам все еще нужно рассмотреть материал в перспективе с помощью Unity, чему вы посвятите оставшуюся часть главы.

Применение ООП в Unity

Если вы достаточно знакомы с объектно-ориентированными языками, то наверняка слышали фразу *«все — объект»*, которую разработчики шепотом передают из уст в уста, словно тайную молитву. Следуя принципам ООП, все в программе должно быть объектом, и игровые объекты в Unity могут реализовывать ваши классы и структуры. Однако это не значит, что все объекты в Unity должны физически находиться на сцене, и потому мы можем использовать наши запрограммированные классы скрыто.

Объекты — это творение класса

В главе 2 мы говорили, как сценарий преобразуется в компонент, когда мы прикрепляем его к `GameObject` в Unity. Подумайте об этом с точки зрения принципа композиции ООП: `GameObject` являются родительскими контейнерами и могут состоять из нескольких компонентов. Это может показаться противоречащим принципу написания одного класса C# в каждом сценарии, но, по правде говоря, это скорее рекомендация, призванная улучшить читабельность, чем реальное требование. Классы можно вкладывать друг в друга, но тогда в коде возникает беспорядок. Однако наличие нескольких компонентов сценария, прикрепленных к одному `GameObject`, может быть очень полезным, особенно при работе с классами-менеджерами или поведением.

Всегда старайтесь создавать классы для самых базовых элементов, а затем с помощью композиции создавайте более крупные и сложные объекты из этих более мелких классов. Легче исправить GameObject, состоящий из небольших взаимозаменяемых компонентов, чем один большой и неуклюжий.

Проверим этот принцип на примере объекта `Main Camera` (рис. 5.13).

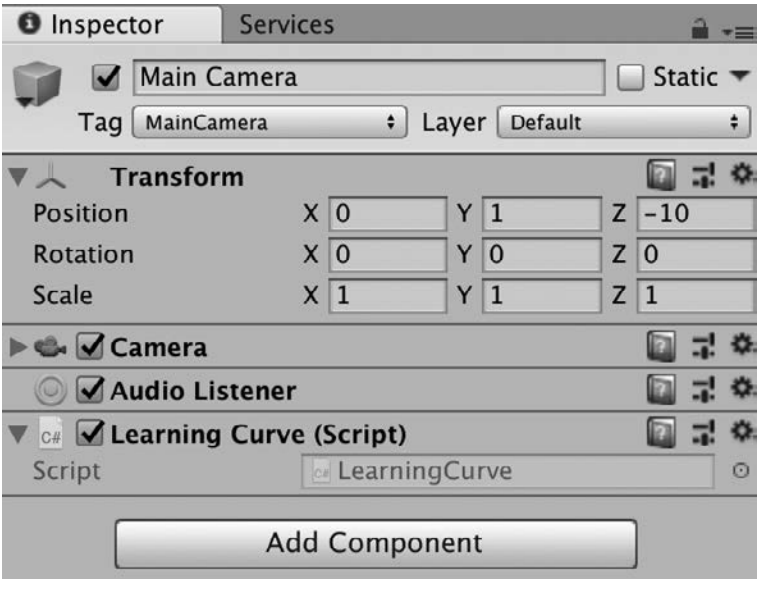


Рис. 5.13

Каждый компонент (`Transform`, `Camera`, `Audio Listener` и сценарий `LearningCurve`) — это класс в Unity. Подобно экземплярам `Character` или `Weapon`, когда мы нажимаем Play, эти компоненты становятся объектами в памяти компьютера вместе с их переменными-членами и методами.

Если бы мы присоединили `LearningCurve` (или любой сценарий или компонент) к 1000 объектам `GameObject` и нажали Play, то движок создал бы и сохранил в памяти 1000 отдельных экземпляров `LearningCurve`.

Мы даже можем создавать наши экземпляры этих компонентов, используя их имя компонента в качестве типа данных. Как и классы,

классы компонентов Unity являются ссылочными типами и могут быть созданы, как и любые другие переменные. Однако поиск и назначение этих компонентов Unity немного отличаются от того, что вы видели до сих пор. Прежде чем продолжить, нам нужно понять, как работают объекты `GameObject`. Об этом в следующем подразделе.

Доступ к компонентам

Теперь, когда мы знаем, как компоненты работают вместе с `GameObject`, нужно выяснить, как получить доступ к их конкретным экземплярам. К счастью для нас, все игровые объекты в Unity наследуются от класса `GameObject`. Это значит, что мы можем использовать их методы-члены, чтобы найти на сцене все необходимое. Есть два способа назначить или получить объекты `GameObject`, которые активны в текущей сцене.

1. Через методы `GetComponent` или `Find` в классе `GameObject`, которые работают с публичными или приватными переменными.
2. Путем перетаскивания самих игровых объектов с панели Project прямо в слоты переменных на панели Inspector. Этот параметр работает только с публичными переменными в C# (или с частными переменными в Unity, отмеченными атрибутом `SerializeField`), поскольку лишь они будут отображаться на панели Inspector.

Вы можете узнать больше об атрибутах и SerializeField в документации Unity, пройдя по ссылке docs.unity3d.com/ScriptReference/SerializeField.html.

Рассмотрим синтаксис первого варианта.

Базовый синтаксис

Использовать метод `GetComponent` довольно просто, но сигнатура этого метода немного отличается от других методов, которые мы видели до сих пор:

```
GameObject.GetComponent<ComponentType>();
```

Нам нужен только тип компонента, который мы ищем, и класс GameObject вернет компонент при его наличии и null в случае отсутствия. Существуют и другие варианты метода GetComponent, но этот самый простой, поскольку нам не нужно знать особенности класса GameObject, который мы ищем. Это метод generic, который мы обсудим далее, в главе 11. Но пока просто поработаем с преобразованием камеры.

Время действовать. Получаем доступ к текущему компоненту преобразования

Поскольку сценарий LearningCurve уже прикреплен к объекту Main Camera, возьмем его компонент Transform и сохраним его в публичной переменной.

1. Добавьте новую публичную переменную типа Transform, называемую camTransform, в сценарий LearningCurve:

```
private Transform camTransform;
```

2. Инициализируйте camTransform в методе Start с помощью метода GetComponent из класса GameObject:
   - используйте ключевое слово this, поскольку LearningCurve прикреплен к тому же компоненту GameObject, что и компонент Transform.
3. Получите доступ к свойству localPosition объекта camTransform и выведите его, используя точечную нотацию:

```
void Start()
{
    camTransform = this.GetComponent<Transform>();
    Debug.Log(camTransform.localPosition);
}
```

Мы добавили неинициализированную переменную private Transform в начало сценария LearningCurve и инициализировали ее с помощью метода GetComponent внутри метода Start. Метод GetComponent находит компонент Transform, прикрепленный к этому компоненту GameObject, и возвращает его camTransform. Теперь, когда camTransform хранит объект Transform, у нас есть доступ ко всем его свойствам и методам класса, включая localPosition (рис. 5.14).

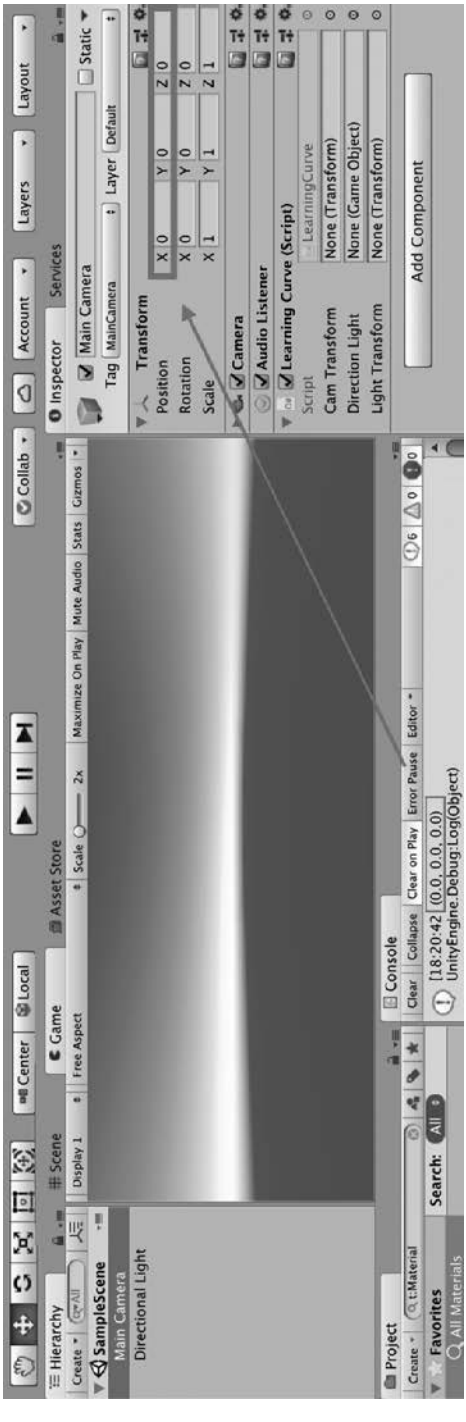

Hierarchy
Create
SampleScene
Main Camera
Directional Light
Scene
Game
Asset Store
Center
Local
Collab
Account
Layers
Layout
Display 1
Free Aspect
Scale
2x
Maximize On Play
Mute Audio
Stats
Gizmos
Inspector
Services
Main Camera
Static
Tag
MainCamera
Layer
Default
Transform
Position
Rotation
Scale
Camera
Audio Listener
Learning Curve (Script)
Script
LearningCurve
Cam Transform
None (Transform)
Direction Light
None (Game Object)
Light Transform
None (Transform)
Add Component
Project
Create
t:Material
Favorites
All Materials
Search:
All
Console
Clear
Collapse
Clear on Play
Error Pause
Editor
[18:20:42] (0.0, 0.0, 0.0)
UnityEngine.Debug:Log(Object)

Рис. 5.14

Метод GetComponent отлично подходит для быстрого получения компонентов, но имеет доступ только к компонентам объекта GameObject, к которому прикреплен вызывающий сценарий. Например, если мы используем метод GetComponent из сценария LearningCurve, прикрепленного к Main Camera, то сможем получить доступ лишь к компонентам Transform, Camera и Audio Listener.

Если мы хотим ссылаться на компонент в отдельном GameObject, таком как Directional Light, то нам нужно сначала получить ссылку на объект, используя метод Find. Для этого потребуется только имя GameObject, и Unity вернет соответствующий GameObject, который мы будем хранить или применять.

Для справки, имя каждого GameObject можно найти в верхней части панели Inspector, когда этот объект выбран (рис. 5.15).

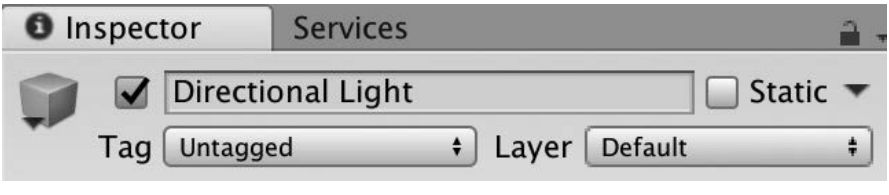


Рис. 5.15

Поиск объектов на сцене в Unity невероятно важен, поэтому вам нужно попрактиковаться. Возьмем объекты, с которыми нам нужно работать, и потренируемся в поиске и назначении их компонентов.

Время действовать. Ищем компоненты на разных объектах

Попробуем использовать метод Find и возьмем объект Directional Light из сценария LearningCurve.

1. Добавьте две переменные в LearningCurve под camTransform — одну типа GameObject и одну типа Transform:

```
public GameObject directionLight;
private Transform lightTransform;
```

2. Найдите компонент `Directional Light` по имени и используйте его для инициализации:

```
void Start()
{
    directionLight = GameObject.Find("Directional Light");
}
```

3. Установите значение `lightTransform` для компонента `Transform`, прикрепленного к `directionLight`, и выведите его `localPosition`. Поскольку `directionLight` теперь является `GameObject`, метод `GetComponent` работает отлично:

```
void Start()
{
    directionLight = GameObject.Find("Directional Light");

    lightTransform = directionLight.GetComponent<Transform>();
    Debug.Log(lightTransform.localPosition);
}
```

Перед запуском игры важно понять, что вызовы методов можно объединить в цепочку для сокращения количества шагов кода. Например, мы могли бы инициализировать `lightTransform` в одной строке, объединив `Find` и `GetComponent`, не прибегая к необходимости проходить через `directionLight`:

```
GameObject.Find("Directional Light").GetComponent<Transform>();
```

Предупреждение: длинные строки связанного кода могут привести к потере читабельности и путанице при работе со сложными приложениями. Запомните правило: желательно избегать строк длиннее, чем в этом примере.

Поиск объектов в коде всегда работает, но вы можете и просто перетащить сами объекты на панель Inspector. Ниже я покажу, как это сделать.

Перетаскивание

Теперь, когда мы научились решать многие задачи путем активного применения чистого кода, кратко рассмотрим функции перетаскивания

в Unity. Оно работает намного быстрее, чем использование класса `GameObject` в коде, но иногда при сохранении или экспорте проектов или при обновлении Unity теряет связи между объектами и переменными, созданными таким образом. Если вам нужно быстро назначить несколько переменных, то обязательно задействуйте эту функцию. Но в целом я советую работать с кодом.

Время действовать. Присваиваем переменные в Unity

Подредактируем сценарий `LearningCurve`, чтобы показать, как назначить компонент `GameObject` с помощью перетаскивания.

1. Закомментируйте следующую строку кода, в которой мы использовали метод `GameObject.Find()` для получения и назначения объекта `Directional Light` переменной `directionLight`:

```
//directionLight = GameObject.Find("Directional Light");
```

2. Выберите объект `Main Camera`, перетащите `Directional Light` в поле `Direction Light` в компоненте `LearningCurve` и нажмите кнопку Play (рис. 5.16).

Объект `Directional Light` теперь назначен переменной `directionLight`. Никакой код не использовался, поскольку Unity присвоил переменную у себя внутри, не внося изменений в класс `LearningCurve`.

Важно понимать пару вещей о назначении переменных с помощью перетаскивания или метода GameObject.Find(). Во-первых, метод Find() работает немного медленнее, что создает проблемы с производительностью, если вы вызываете метод несколько раз в нескольких сценариях. Во-вторых, вы должны быть уверены, что все ваши GameObject имеют уникальные имена. В противном случае возможны неприятные ошибки, если у вас есть несколько объектов с одинаковыми именами или меняются сами имена объектов.

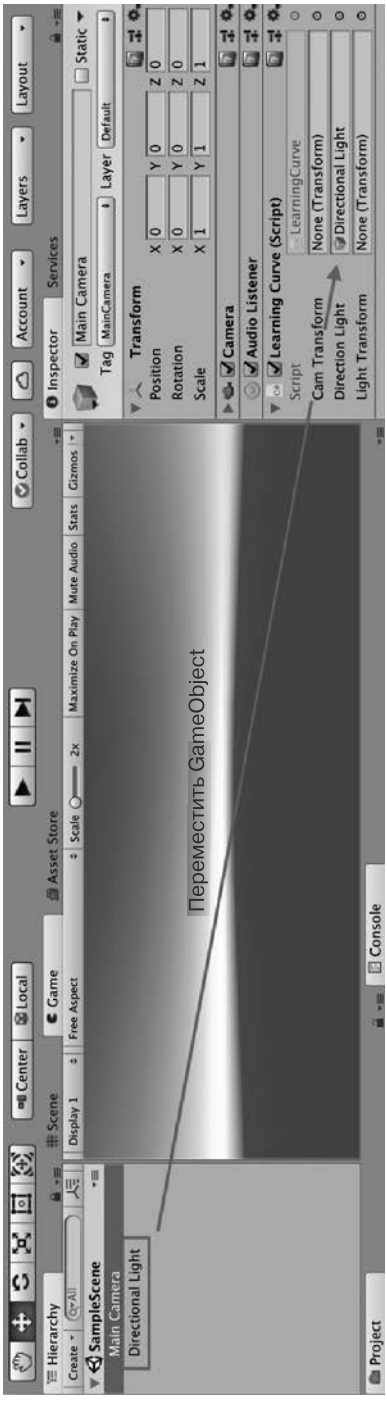

Center
Local
Collab
Account
Layers
Layout
Hierarchy
Scene
Game
Asset Store
Inspector
Services
Create
SampleScene
Main Camera
Directional Light
Display 1
Free Aspect
Scale
2x
Maximize On Play
Mute Audio
Stats
Gizmos
Переместить GameObject
Main Camera
Static
Tag
MainCamera
Layer
Default
Transform
Position
Rotation
Scale
Camera
Audio Listener
Learning Curve (Script)
Script
LearningCurve
Cam Transform
None (Transform)
Direction Light
Directional Light
Light Transform
None (Transform)
Project
Console

Рис. 5.16

Подведем итоги

Наше путешествие в классы, структуры и ООП знаменует собой конец первой части, посвященной основам C#. Вы узнали, как объявлять собственные классы и структуры — каркасы для любого приложения или игры, которые вы когда-либо будете создавать. Кроме того, мы определили различия между этими объектами с точки зрения доступа и передачи, а также разобрались, как они связаны с ООП. Наконец, вы познакомились с основными понятиями ООП: созданием классов с помощью наследования, композиции и полиморфизма.

Поиск связанных данных и действий, создание шаблонов для придания им формы и использование экземпляров для создания взаимодействий — все это создает прочную основу для подхода к любой программе или игре. Добавьте сюда возможность доступа к компонентам, и вы получите задатки разработчика Unity.

В следующей главе мы рассмотрим основы разработки игр и сценариев поведения объектов непосредственно в Unity. Мы начнем с конкретизации требований к простой приключенческой игре с открытым миром, поработаем с игровыми объектами на сцене и затем подготовим среду для наших персонажей.

Контрольные вопросы. Все об ООП

1. Какой метод обрабатывает логику инициализации внутри класса?
2. Как передаются структуры, которые относятся к типу «значение»?
3. Каковы основные концепции ООП?
4. Какой метод класса `GameObject` вы бы использовали для поиска компонента на объекте вызывающего класса?

6 Погружение в Unity

Создание игры — нечто большее, чем просто имитация действий в коде. Дизайн, сюжет, окружение, освещение и анимация — все это играет важную роль в создании мира для игрока. Игра — прежде всего опыт, который не может дать один только код.

За последнее десятилетие движок Unity вышел на лидирующие позиции в разработке игр и сегодня предлагает передовые инструменты как программистам, так и тем, кто не занимается программированием. Анимация и эффекты, звук, дизайн среды и др. — все это доступно прямо из редактора Unity и все это можно создавать, вообще не прикасаясь к коду. К этим вопросам мы еще вернемся, когда будем определять требования, среду и игровую механику нашей игры. Однако для начала нам нужно познакомиться с основами игрового дизайна.

Теория игрового дизайна — серьезная область, и изучение всех ее секретов может занять всю жизнь. Здесь мы познакомимся только с основами, а остальное зависит от вас! Эта глава подготовит нас к остальной части книги, и мы рассмотрим здесь такие темы, как:

- основы игрового дизайна;
- создание уровня;
- основы игровых объектов и префабов;
- основы работы со светом;
- анимация в Unity;
- интеграция системы частиц.

Основы игрового дизайна

Прежде чем приступить к любому игровому проекту, важно иметь план того, что вы хотите получить в итоге. Иногда в виде идеи все выглядит у вас в голове просто идеально, но стоит вам начать создавать классы персонажей или среды, кажется, что все отклоняется от вашего первоначального намерения. А вот дизайн игры позволяет вам планировать следующие точки соприкосновения:

- **концепцию** — общую идею и дизайн игры, включая ее жанр и стиль игры;
- **основную механику** — игровые функции или взаимодействия, которые персонаж может выполнять в игре. Это могут быть прыжки, стрельба, решение головоломок, управление транспортом и т. п.;
- **схему управления** — карту кнопок и/или клавиш, которые позволяют игрокам управлять своим персонажем, взаимодействовать с окружающей средой и другими выполняемыми действиями;
- **сюжет** — основное повествование, закладывающее основу игры, заставляющее игроков сопереживать и проникаться миром, в котором они играют;
- **художественный стиль** — общий вид и атмосферу игры, одинаковые для всего, от персонажей и рисунков меню до уровней и окружающей среды;
- **условия выигрыша и проигрыша** — правила, определяющие, как выиграть или проиграть игру, обычно состоящие из целей или задач, которые несут в себе возможность неудачи.

И это далеко не полный список всего, что входит в разработку игры. Но и этих пунктов достаточно, чтобы начать конкретизировать документацию по игровому дизайну. И именно этим мы и займемся!

Документация игрового дизайна

Если вы поищете в Google документы по дизайну игр, то найдете бесчисленное множество шаблонов, правил форматирования и рекомендаций по содержанию, которые могут заставить новоиспеченного про-

граммиста бросить эту затею. Все дело в том, что проектные документы адаптированы под конкретную команду или компанию, которая их написала, что упрощает их разработку куда больше, чем может показаться после поисков в Интернете.

Существует три типа документации по дизайну.

- **Документ дизайна игры** (Game Design Document, GDD) хранит все, от механики игры до ее атмосферы, сюжета и опыта, который она пытается создать. В зависимости от конкретной игры этот документ может состоять из нескольких или нескольких сотен страниц.
- **Документ технического дизайна** (Technical Design Document, TDD) рассматривает все технические аспекты игры, от оборудования, на котором она будет работать, до устройства классов и архитектуры программы. Как и в GDD, объем этого документа зависит от проекта.
- **Брошюра** (One-Page) обычно используется в маркетинговых или рекламных целях. Это краткая выжимка самого важного о вашей игре. Как следует из названия, брошюра должна занимать только одну страницу.

Не существует правильного или неправильного способа создания GDD, поэтому в написании документации можно дать свободу творчеству. Вы можете добавлять изображения и справочные материалы, которые вас вдохновляют, творчески подходить к созданию макета и т. д. — именно здесь вы можете изложить свое видение так, как хочется.

Игра, над которой мы будем работать до конца книги, довольно проста и не потребует написания сложных документов наподобие GDD или TDD. Вместо этого мы создадим брошюру, в которой изложим цели нашего проекта и некую справочную информацию.

Брошюра о Hero Born

Чтобы у нас был ориентир на пути, я составил простой документ, в котором будут изложены основные аспекты прототипа игры. Прочитайте

его, прежде чем двигаться дальше, и попытайтесь представить, как некоторые из понятий программирования, изученные нами ранее, будут применяться в этой игре на практике.

- **Концепция.** Основная задача игры — скрытно избегать противников и собирать предметы, восстанавливающие здоровье. В игре также будут элементы шутера от первого лица.
- **Геймплей.** Основная механика игры — следить за полем зрения врагов, чтобы предугадывать их действия, и собирать нужные предметы.

  В бою игрок будет стрелять в противников, что автоматически вызывает срабатывания ответной реакции.
- **Интерфейс.** Схема управления — движение клавишами W, A, S и D или стрелками, а управление камерой — с помощью мыши. Стрельба осуществляется путем нажатия клавиши пробела, а сбор предметов реализуется через столкновения.

  В игре будет простой интерфейс, на котором будет отображаться количество собранных предметов и количество имеющихся боеприпасов, а также шкала здоровья.
- **Художественный стиль.** Уровень и персонаж будут выполнены в виде примитивных объектов в целях ускорения разработки. Позже при необходимости модели и уровень можно заменить на более совершенные, добавить элементы среды.

Теперь, имея высокоуровневое представление об устройстве игры, вы готовы приступить к созданию прототипа уровня для размещения игрового опыта.

Создание уровня

При создании игровых уровней всегда полезно попытаться взглянуть на игру с точки зрения игрока. Какой игрок должен увидеть окружающую среду, как взаимодействовать с ней? Какие ощущения он должен по-

лучить, прогуливаясь по уровню? Вы сами создаете мир, в котором существует ваша игра, поэтому будьте последовательны.

В Unity вы можете создавать «уличное» окружение с помощью инструмента Terrain, а помещения — с помощью геометрических примитивов или их сочетаний. Вы даже можете импортировать 3D-модели из других программ, таких как Blender, и использовать их в качестве объектов в сценах.

У Unity есть отличное вводное руководство по работе с инструментом Terrain (docs.unity3d.com/ru/current/Manual/script-Terrain.html). Если вы решите следовать ему, то в магазине Unity Asset Store есть замечательный бесплатный ресурс под названием Terrain Toolkit 2017 (assetstore.unity.com/packages/tools/terrain/terrain-toolkit-2017-83490).

В игре Hero Born мы создадим простую закрытую арену, которую будет легко обойти, и разместим на ней парочку укрытий, где можно будет спрятаться. Соберем мы нашу арену из примитивов — базовых объектов, имеющихся в Unity. С примитивами легко работать, создавать их, масштабировать и размещать в сцене.

Создание примитивов

Глядя на игры, в которые вы играете, вы, вероятно, задаетесь вопросом, как у разработчиков получается создавать настолько реалистичные модели и объекты, выглядящие так, словно можно дотянуться до экрана и потрогать их руками. К счастью, в Unity есть набор примитивных `GameObject`, из которых можно быстро собрать нужный прототип. Они не будут супермодными, и у них не будет высокого разрешения, но они отлично подходят для изучения основ или на случай, когда в вашей команде разработчиков нет 3D-художника.

Открыв панель Hierarchy и выбрав команду Create ▸ 3D-object, вы увидите все доступные параметры, и около половины из них являются примитивами или обычными геометрическими объектами (рис. 6.1).

Другие варианты 3D-объектов, такие как Terrain, Wind Zone и Tree, слишком сложны для нас на данный момент, но вы все же можете поэкспериментировать с ними, если вдруг будет интересно. Прежде чем мы запрыгнем слишком далеко вперед, арене понадобится пол, поэтому создадим его.

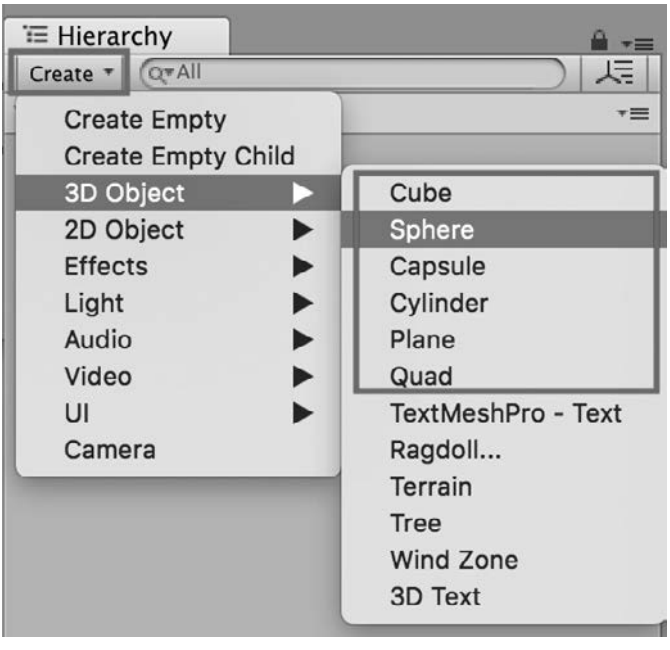


Рис. 6.1

Время действовать. Создаем плоскость

Обычно ходить легче, когда под вами есть пол, поэтому начнем с создания плоскости, выполнив следующие шаги.

1. На панели Hierarchy выберите Create ▸ 3D Object ▸ Plane.
2. Переименуйте появившийся объект в `Ground` на панели Inspector:
   - измените свойство `Scale` по осям `X`, `Y` и `Z` на `3` (рис. 6.2).
3. Если освещение в вашей сцене выглядит тусклее или отличается от приведенного на рис. 6.2, увеличьте значение `Intensity` компонента `Directional Light` следующим образом (рис. 6.3).

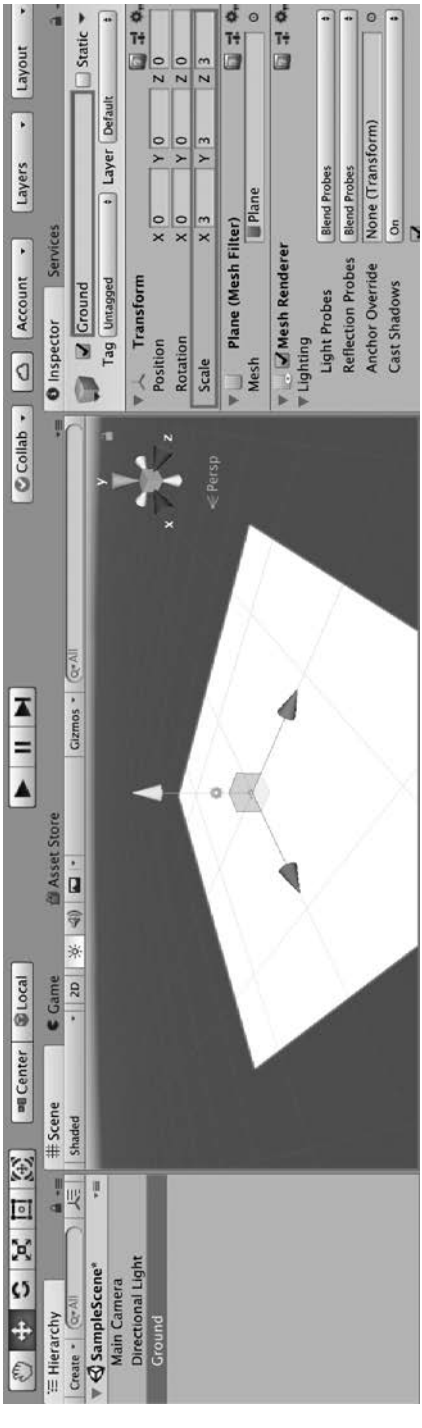

Hierarchy
Create
SampleScene*
Main Camera
Directional Light
Ground
Scene
Shaded
2D
Center
Local
Game
Asset Store
Gizmos
Persp
Collab
Account
Layers
Layout
Inspector
Services
Ground
Static
Tag
Untagged
Layer
Default
Transform
Position
Rotation
Scale
Plane (Mesh Filter)
Mesh
Plane
Mesh Renderer
Lighting
Light Probes
Reflection Probes
Anchor Override
Cast Shadows
Blend Probes
None (Transform)
On

Рис. 6.2

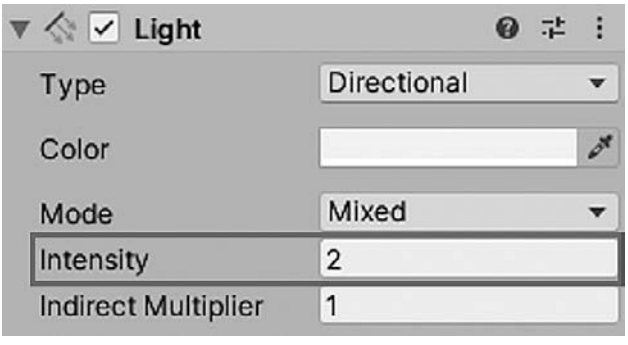


Рис. 6.3

Мы создали плоскость и увеличили ее размер, чтобы у нашего будущего персонажа было больше места для прогулок. Она будет работать как трехмерный объект с реальной физикой, а это значит, что другие объекты не будут падать сквозь нее. Позже, в главе 7, мы подробнее поговорим о системе физики в Unity и о том, как она работает. А пока добавим нашему мышлению третье измерение.

Трехмерное мышление

Теперь, когда на сцене появился первый объект, мы можем поговорить о трехмерном пространстве, а именно о том, как в таком пространстве работают параметры положения, вращения и масштаба объекта. Если вы вспомните школьную алгебру, то там наверняка были графики с системой координат x и y. Чтобы поставить точку на графике, у вас должны быть значения x и y.

Unity поддерживает разработку как 2D-, так и 3D-игр, и если бы мы делали 2D-игру, то могли бы и не продолжать. Однако при работе с трехмерным пространством в Unity появляется дополнительная ось, называемая осью z. Это ось глубины, или перспективы, и она придает нашему пространству и объектам трехмерность.

Поначалу вы можете запутаться, но в Unity есть несколько хороших наглядных пособий, которые помогут вам во всем разобраться. В правом верхнем углу панели Scene есть замысловатый значок с осями x, y и z,

отмеченными красным, зеленым и синим соответственно. У всех `GameObject` на сцене тоже есть свои оси, которые появляются, когда их выбирают (рис. 6.4).

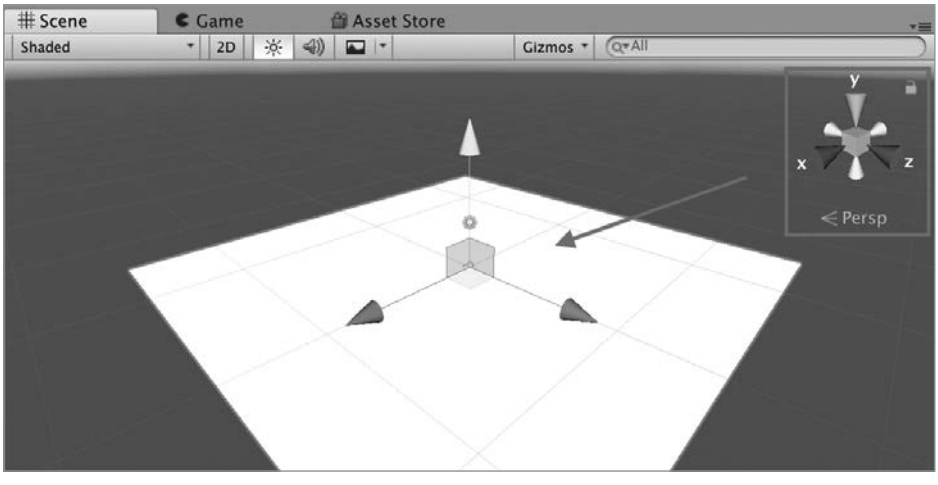


Рис. 6.4

Оси всегда показывают текущую ориентацию сцены и объектов, размещенных внутри нее. Щелчок на любой из этих цветных осей переключит ориентацию сцены на выбранную ось. Попробуйте сами нажимать стрелки, чтобы привыкнуть к переключению ракурсов.

Если вы посмотрите на компонент `Transform` объекта `Ground` на рис. 6.5, то увидите, что положение, поворот и масштаб объекта (свойства `Transform`, `Rotation` и `Scale`) определяются этими тремя осями. Положение определяет, где объект находится на сцене, вращение — угол его наклона, а масштаб — его размер.

Это подводит нас к интересному вопросу: а от каких точек отсчитываются значения положения, поворота и масштаба, которые мы устанавливаем? Ответ зависит от того, какое относительное пространство мы используем. В Unity это либо `World`, либо `Local`:

- в мировом пространстве (`World`) некая заданная исходная точка сцены используется в качестве постоянной точки отсчета для всех объектов.

В Unity этой исходной точкой является точка (0, 0, 0), именно она изначально указана в компоненте `Transform` объекта `Ground`;

- в локальном пространстве (`Local`) в качестве точки отсчета используется преобразование родительского объекта, что меняет перспективу сцены при вращении.

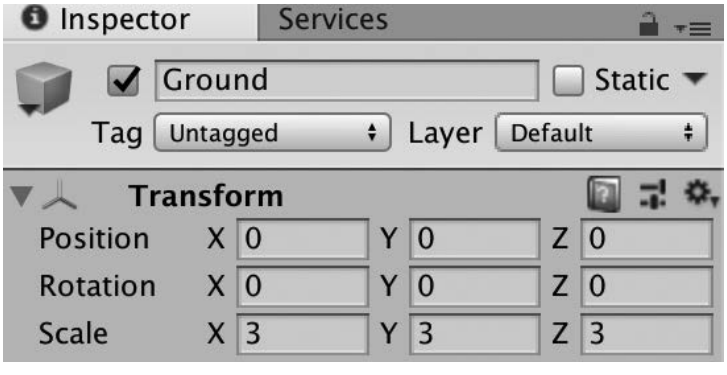


Рис. 6.5

Обе ориентации полезны в разных ситуациях, о которых мы поговорим позже в данной главе, когда начнем создавать остальную часть арены. Сейчас пол нашей арены немного скучный. Это можно изменить с помощью материала.

Материалы

Наш пол сейчас не очень интересен, но мы можем использовать материалы, чтобы вдохнуть в уровень немного жизни. Материалы контролируют рендеринг игровых объектов на сцене, что определяется шейдером материала. Именно шейдеры отвечают за объединение данных освещения и текстур во внешний вид материала.

У каждого `GameObject` есть компоненты `Material` и `Shader` по умолчанию (на рис. 6.6 представлен скриншот панели `Inspector`), которые задают стандартный белый цвет.

Чтобы изменить цвет объекта, нужно создать материал и перетащить его на объект, который мы хотим изменить. Помните, что в Unity все

является объектом, и с материалами та же история. Материалы можно использовать на любом количестве GameObject, но при этом любое изменение конкретного Material также повлияет на все объекты, к которым он прикреплен. Если бы у нас на сцене было несколько объектов — противников красного цвета и мы изменили бы этот цвет на синий, то все наши враги стали бы синими.

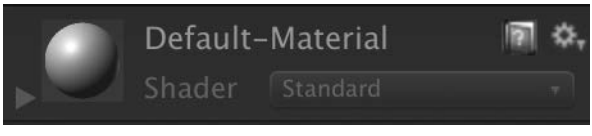


Рис. 6.6

Кстати, синий привлекает внимание, поэтому изменим цвет арены, чтобы он тоже был красивым.

Время действовать. Меняем цвета арены

Создадим новый материал, чтобы покрасить нашу базовую плоскость из тускло-белого в темный и яркий синий.

1. Создайте на панели Project новую папку и назовите ее Materials.
2. В папке Materials выберите команду Create ▸ Material и назовите материал Ground_Mat.
3. Нажмите поле цвета рядом со свойством Albedo и выберите цвет на свой вкус из всплывающего окна Color Picker, а затем закройте его.
4. Перетащите объект Ground_Mat на объект Ground на панели Hierarchy (рис. 6.7).

Созданный вами новый материал теперь является ассетом (asset) проекта. Перетаскивание Ground_Mat на объект Ground изменило цвет плоскости; это значит, что теперь любые изменения материала Ground_Mat будут отражаться на Ground (рис. 6.8).

Пол — это наш холст. В трехмерном пространстве на данной поверхности могут стоять и другие трехмерные объекты. Вам предстоит наполнить сцену веселыми и интересными препятствиями для будущих игроков.

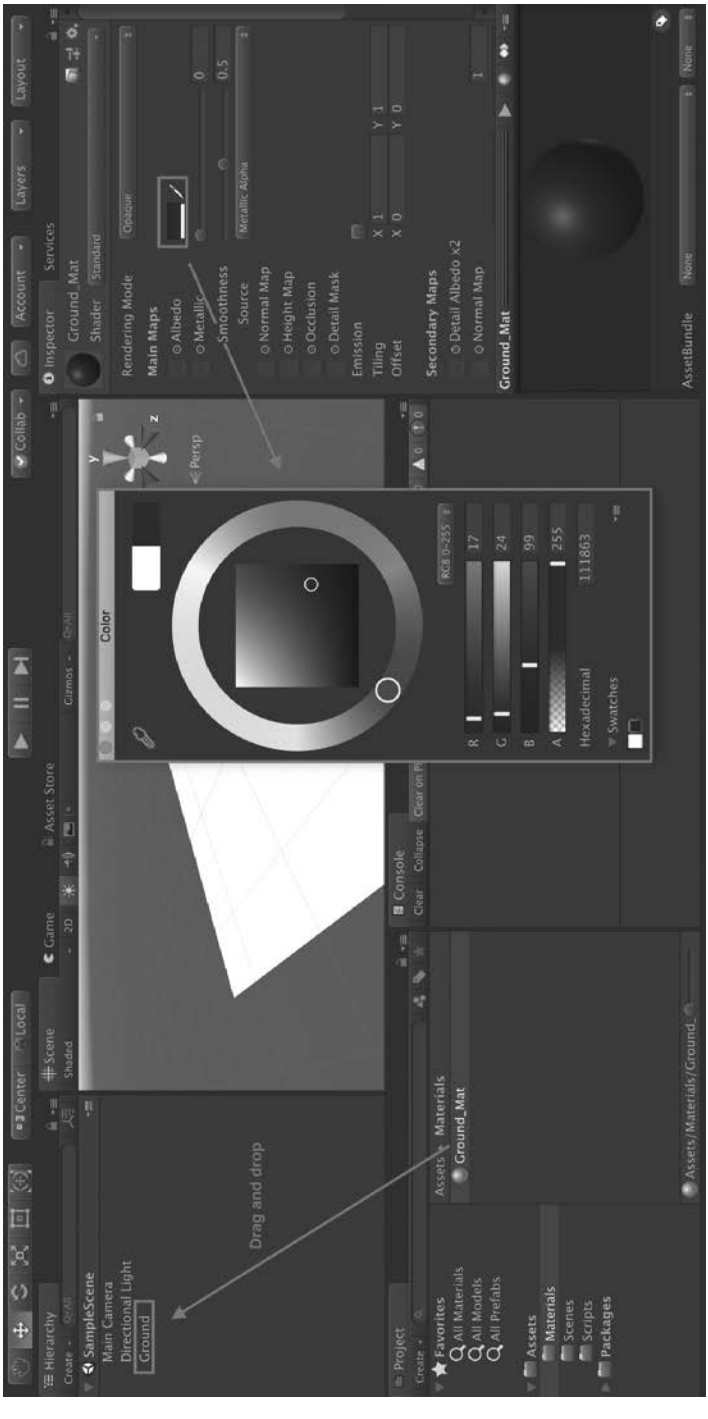

Hierarchy
Create
SampleScene
Main Camera
Directional Light
Ground
Drag and drop
Center
Local
Scene
Shaded
Game
2D
Asset Store
Gizmos
Collab
Persp
Color
RGB 0-255
R 17
G 24
B 99
A 255
Hexadecimal 111863
Swatches
Project
Create
Favorites
All Materials
All Models
All Prefabs
Assets
Materials
Scenes
Scripts
Packages
Assets > Materials
Ground_Mat
Assets/Materials/Ground...
Console
Clear
Collapse
Inspector
Services
Ground_Mat
Shader Standard
Rendering Mode Opaque
Main Maps
Albedo
Metallic
Smoothness 0
Source Metallic Alpha
Normal Map
Height Map
Occlusion
Detail Mask
Emission
Tiling X 1 Y 1
Offset X 0 Y 0
Secondary Maps
Detail Albedo x2
Normal Map 1
Ground_Mat
AssetBundle None None
Account
Layers
Layout

Рис. 6.7

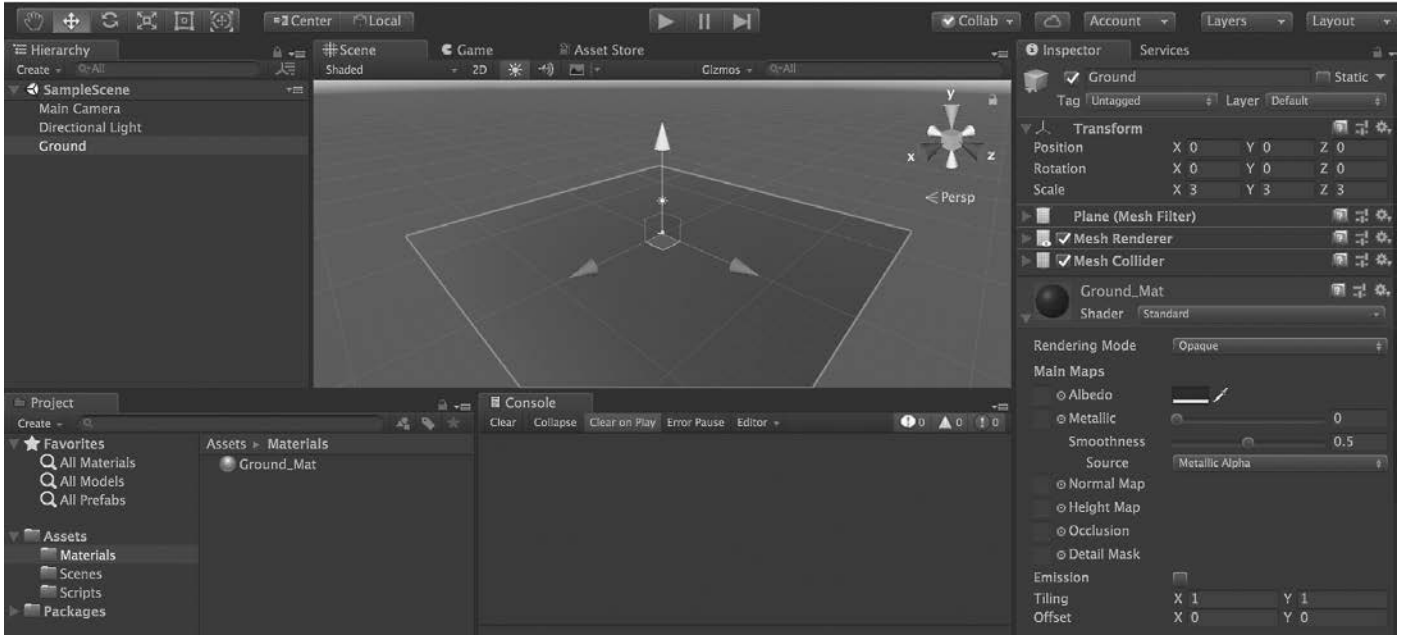

Center
Local
Collab
Account
Layers
Layout
Hierarchy
Create
SampleScene
Main Camera
Directional Light
Ground
Scene
Game
Asset Store
Shaded
2D
Gizmos
Persp
Inspector
Services
Ground
Static
Tag
Untagged
Layer
Default
Transform
Position
Rotation
Scale
Plane (Mesh Filter)
Mesh Renderer
Mesh Collider
Ground_Mat
Shader
Standard
Rendering Mode
Opaque
Main Maps
Albedo
Metallic
Smoothness
Source
Metallic Alpha
Normal Map
Height Map
Occlusion
Detail Mask
Emission
Tiling
Offset
Project
Favorites
All Materials
All Models
All Prefabs
Assets
Materials
Scenes
Scripts
Packages
Assets ▸ Materials
Ground_Mat
Console
Clear
Collapse
Clear on Play
Error Pause
Editor

Рис. 6.8

White-boxing

White box — термин, обозначающий размещение идей в виде пустых заполнителей, при этом обычно планируется заменять их готовыми ассетами позже. В дизайне уровней практика White box заключается в заполнении среды примитивными игровыми объектами, чтобы получить общее представление о том, как сцена должна выглядеть по вашему мнению. Это отличный способ начать работу, особенно на этапе создания прототипа вашей игры.

Прежде чем погрузиться в Unity, я хотел бы начать с простого наброска базовой компоновки и положения уровня. Это даст нам направление движения и поможет быстрее создать среду. На рис. 6.9 показана арена, которую я представляю, с приподнятой платформой посередине, пандусами для доступа на нее и небольшими башнями по углам.

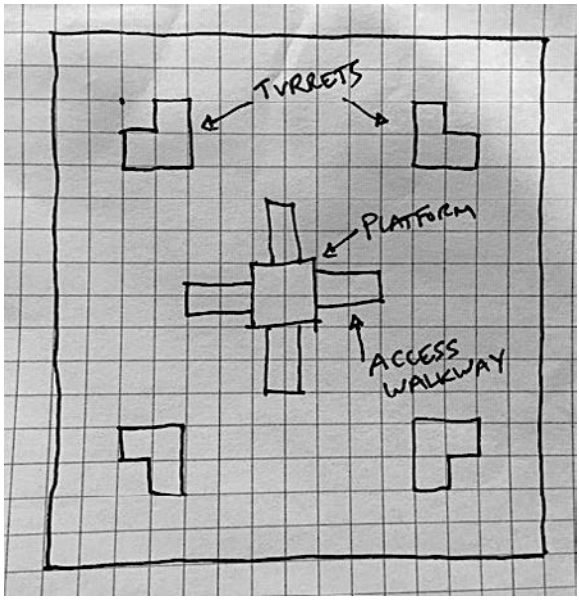


Рис. 6.9

Не волнуйтесь, если вы не художник, — я тоже. Важно записывать свои идеи на бумаге, чтобы закрепить их в своей голове и ответить на возникающие вопросы, прежде чем приступить к работе с Unity.

Перед тем как полностью погрузиться в работу и запустить этот набросок в производство, вам необходимо ознакомиться с несколькими ярлыками в редакторе Unity, которые упростят работу.

Инструменты редактора

Обсуждая интерфейс Unity в главе 1, мы бегло рассмотрели ряд функций панели инструментов, которые нам снова следует вспомнить, чтобы знать, как эффективно управлять игровыми объектами (рис. 6.10).

Рис. 6.10

Рассмотрим различные инструменты, размещенные на панели инструментов Toolbar и показанные на рис. 6.10.

1. Hand: позволяет вам изменять свое положение на сцене и поворачивать камеру.
2. Move: позволяет перемещать объекты по осям *x*, *y* и *z*, «держась» за их соответствующие стрелки.
3. Rotate: позволяет настраивать поворот объекта, поворачивая или перетаскивая соответствующие маркеры.
4. Scale: позволяет вам изменять масштаб объекта, перетаскивая его определенные оси.
5. Rect Transform: функции перемещения, поворота и масштабирования, собранные в одном месте.
6. Move, Rotate and Scale: дает доступ к положению, повороту и масштабированию объекта одновременно.

Пользовательские инструменты редактора позволяют вам получить доступ к любым созданным вами инструментам редактора. Об этом пока можно забыть, поскольку это выходит за рамки наших возможностей. Если вы хотите узнать больше, то обратитесь к документации (docs.unity3d.com/2020.1/Documentation/ScriptReference/EditorTools.EditorTool.html).

Дополнительную информацию о навигации и позиционировании объектов на панели Scene можно найти по адресу docs.unity3d.com/ru/current/Manual/PositioningGameObjects.html.

Поворот камеры и перемещение по сцене можно выполнять с помощью альтернативных методов, не прибегая к кнопкам редактора Unity:

- чтобы повернуть камеру, удерживайте правую кнопку мыши;
- чтобы дополнительно перемещаться в этом режиме, продолжайте удерживать правую кнопку мыши и используйте клавиши `W`, `A`, `S` и `D` для перемещения вперед, назад, влево и вправо соответственно;
- нажмите клавишу `F`, чтобы увеличить масштаб и сфокусироваться на выбранном `GameObject`.

Этот вид навигации по сцене более известен как режим пролета, и когда я буду просить вас сфокусироваться на объекте или перейти к определенному объекту либо точке обзора, используйте перечисленные выше функции.

Перемещение по сцене и поиск ракурса иногда сам по себе может быть непростой задачей, но этот навык нарабатывается с практикой. Для получения более подробного списка функций навигации по сцене пройдите по ссылке docs.unity3d.com/ru/current/Manual/SceneViewNavigation.html.

Несмотря на то что базовая плоскость не позволяет нашему персонажу провалиться сквозь нее, шагнуть с обрыва ничто не мешает. Ваша задача — поставить на арене стену, чтобы область передвижения игрока была ограничена.

Испытание героя. Укладываем гипсокартон

Используя примитивные кубы и панель инструментов, разместите на уровне четыре стены с помощью инструмента Move, Rotate and Scale, чтобы оградить главную арену. В качестве ориентира пригодится снимок экрана, приведенный далее (рис. 6.11).

Начиная с этой главы, я не привожу точные значения положения стены, поворота или масштаба, поскольку хочу, чтобы вы поэкспериментировали с инструментами редактора Unity. Единственный способ обрести здесь свободу — это нырнуть в омут с головой.

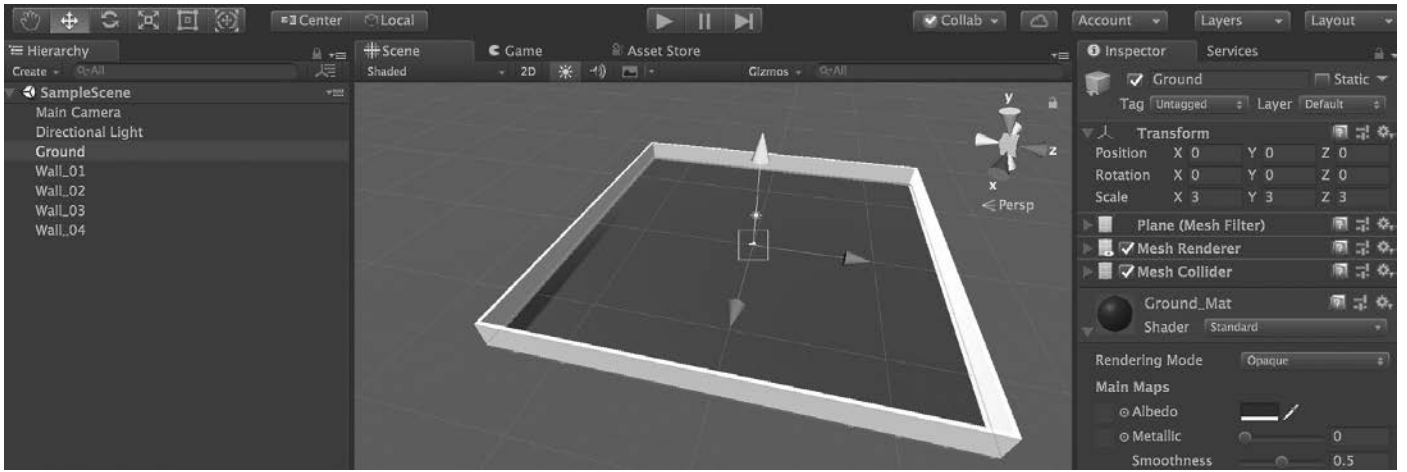

Center
Local
Collab
Account
Layers
Layout
Hierarchy
Create
SampleScene
Main Camera
Directional Light
Ground
Wall_01
Wall_02
Wall_03
Wall_04
Scene
Game
Asset Store
Shaded
2D
Gizmos
y
z
x
Persp
Inspector
Services
Ground
Static
Tag
Untagged
Layer
Default
Transform
Position
Rotation
Scale
Plane (Mesh Filter)
Mesh Renderer
Mesh Collider
Ground_Mat
Shader
Standard
Rendering Mode
Opaque
Main Maps
Albedo
Metallic
Smoothness

Рис. 6.11

Пришлось повозиться, но зато арена начинает обретать форму! Прежде чем мы перейдем к добавлению препятствий и платформ, нужно выработать привычку наводить порядок в иерархии объектов. Мы поговорим о том, как это делать, ниже.

Поддержание порядка в иерархии

Обычно такие советы приводят в конце раздела, но поддержание порядка в иерархии проекта настолько важно, что я решил выделить этому вопросу отдельный пункт. В идеале все связанные `GameObject` должны находиться под одним родительским объектом. Прямо сейчас это не столь важно, поскольку у нас на сцене всего несколько объектов, но, когда в большом проекте количество объектов достигает сотен, начинаются проблемы.

Самый простой способ сохранить чистоту иерархии — хранить связанные объекты в родительском объекте, как если бы вы хранили файлы внутри папки на рабочем столе.

Время действовать. Используем пустые предметы

На нашем уровне есть несколько объектов, которые уже можно было бы организовать, и в Unity данный процесс упрощается путем создания пустых `GameObject`. Пустой объект — идеальный контейнер для хранения связанных групп объектов, поскольку к нему в комплекте не идет ничего. Это просто пустая оболочка.

Возьмем базовую плоскость и четыре стены и сгруппируем их под общим пустым `GameObject`.

1. Выберите команду Create ▸ Create Empty на панели Hierarchy и назовите новый объект `Environment`.
2. Перетащите пол и четыре стены в объект `Environment`, чтобы сделать их дочерними объектами.
3. Выберите пустой объект `Environment` и убедитесь, что его позиции по осям `x`, `y` и `z` равны `0` (рис. 6.12).

Объект `Environment` находится на панели Hierarchy как родительский, а содержимое арены — его дочерние объекты. Теперь мы можем рас-

крывать или закрывать раскрывающийся список объектов `Environment` с помощью значка стрелки, благодаря чему на панели Hierarchy будет больше порядка.

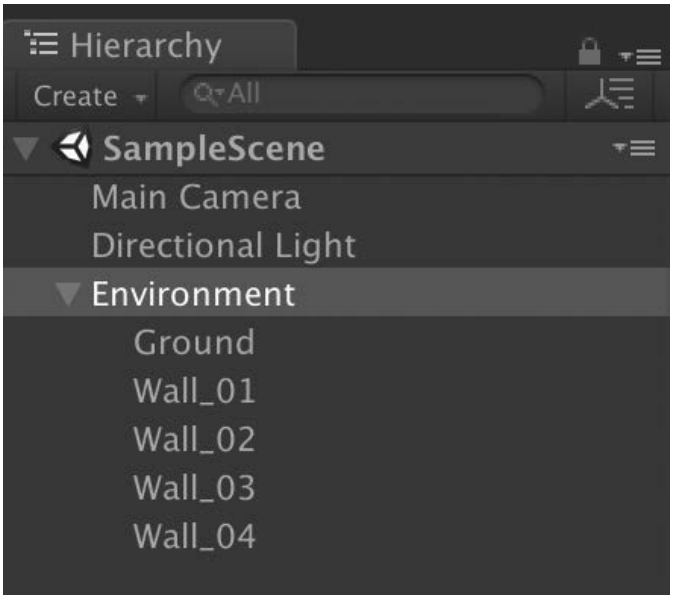


Рис. 6.12

Важно установить позиции `x`, `y` и `z` объекта `Environment` равными `0`, поскольку позиции дочерних объектов теперь отсчитываются относительно родителя.

Работа с префабами

Префабы — один из самых эффективных инструментов, имеющихся в Unity. Они нужны не только при построении уровней, но и при написании сценариев. Для нас префабы — это объекты `GameObject`, которые можно сохранять и повторно использовать с каждым дочерним объектом, компонентом, сценарием C# и настройками без изменений. Будучи созданным, префаб становится похож на класс. Каждая его копия, применяемая на сцене, является отдельным экземпляром этого префаба. Следовательно, любое изменение базового префаба также изменит все активные экземпляры в сцене.

Арена выглядит слишком простой и полностью открыта, что делает ее идеальным местом для тестирования создания и редактирования префабов.

Время действовать. Создаем башню

Поскольку нам нужно создать четыре одинаковые башни в каждом углу арены, они идеально подходят для создания префаба. Так и поступим.

И снова я не указывал никаких точных значений положения, поворота или масштаба, поскольку хочу, чтобы вы сами поближе познакомились с инструментами редактора Unity.

Забегая вперед: когда вы видите перед собой задачу, в которой не указаны конкретные значения положения, поворота или масштаба, это сделано специально, чтобы вы попрактиковались сами.

1. Создайте пустой родительский объект внутри объекта `Environment`, выбрав команду Create ▸ Создайте Empty, и назовите его `Barrier_01`.
2. Создайте два примитива Cube из меню Create, расположив их так, чтобы получить V-образную основу.
3. Создайте еще два примитива Cube и разместите их на концах основания башни (рис. 6.13).

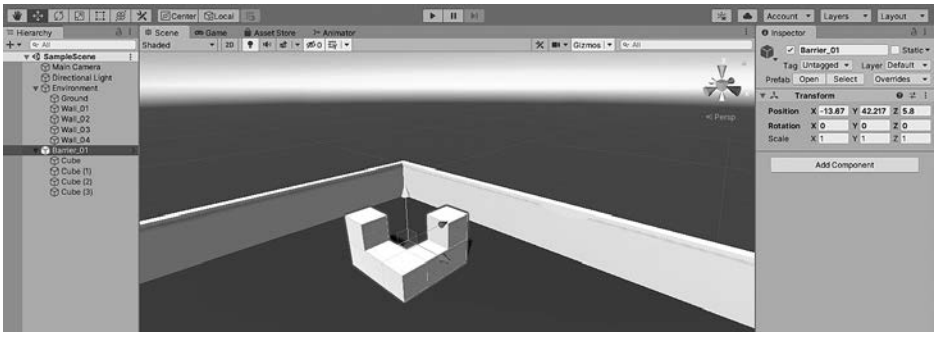


Рис. 6.13

4. Создайте новую папку на панели Project и назовите ее Prefabs. Затем перетащите в нее `Barrier_01` (рис. 6.14).

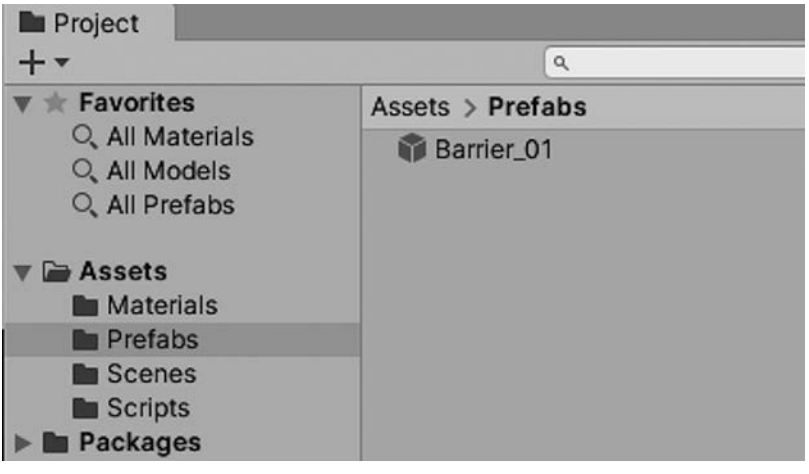


Рис. 6.14

Объект `Barrier_01` и все его дочерние объекты автоматически станут префабами, а это значит, что мы можем повторно использовать их, перетаскивая копии из папки Prefabs или копируя уже имеющийся объект. Объект `Barrier_01` стал синим на панели Hierarchy, что означает изменение его статуса. Кроме того, под его именем на панели Inspector появилось несколько кнопок для работы с префабами (рис. 6.15).

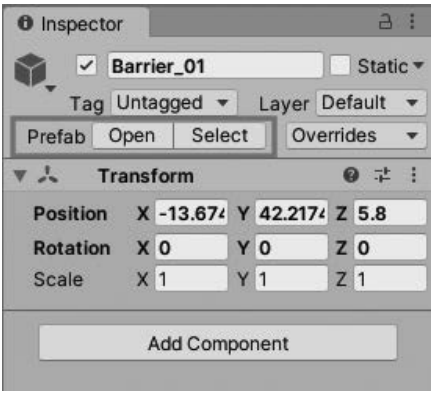


Рис. 6.15

Любые изменения в исходном префабе теперь будут влиять на все его копии в сцене. Поскольку нам нужен третий куб для завершения барьера, обновим и сохраним префаб, чтобы проверить все в действии.

Время действовать. Обновляем префаб

Пока у башни зияет огромный промежуток посередине, что не идеально для прикрытия нашего персонажа, поэтому обновим префаб `Barrier_01`, добавив еще один куб и применив изменения.

1. Создайте примитив Cube и разместите его на пересечении основания башни.
2. Новый примитив `Cube` будет окрашен серым цветом с маленьким значком + рядом с его именем на панели Hierarchy (рис. 6.16). Это значит, что официально он еще не входит в состав префаба.

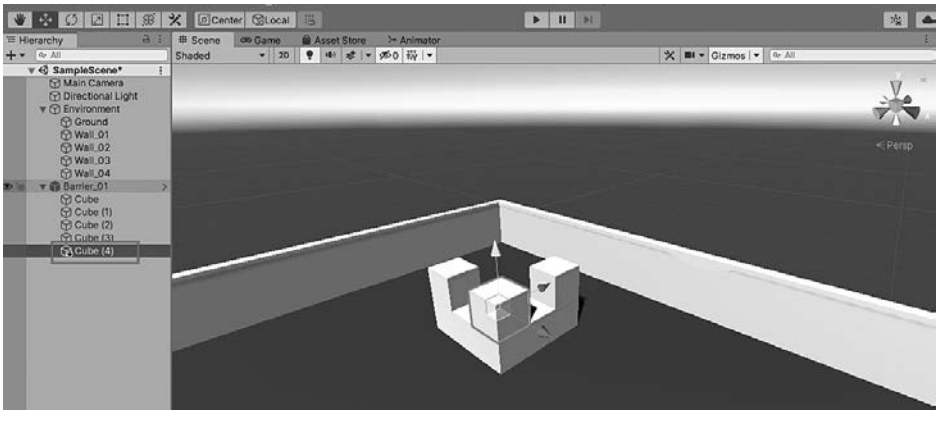


Рис. 6.16

3. Щелкните правой кнопкой мыши по новому примитиву `Cube` и выберите команду Added GameObject ▸ Apply to Prefab 'Barrier_01' (рис. 6.17).

Префаб `Barrier_01` обновится, в нем появится новый куб, и вся иерархия префаба снова должна быть синей. Теперь полученная башня выглядит как на рис. 6.17 или, если вы не боитесь экспериментов, как что-то более креативное. Однако нам нужны такие в каждом углу арены. Пора добавить их!

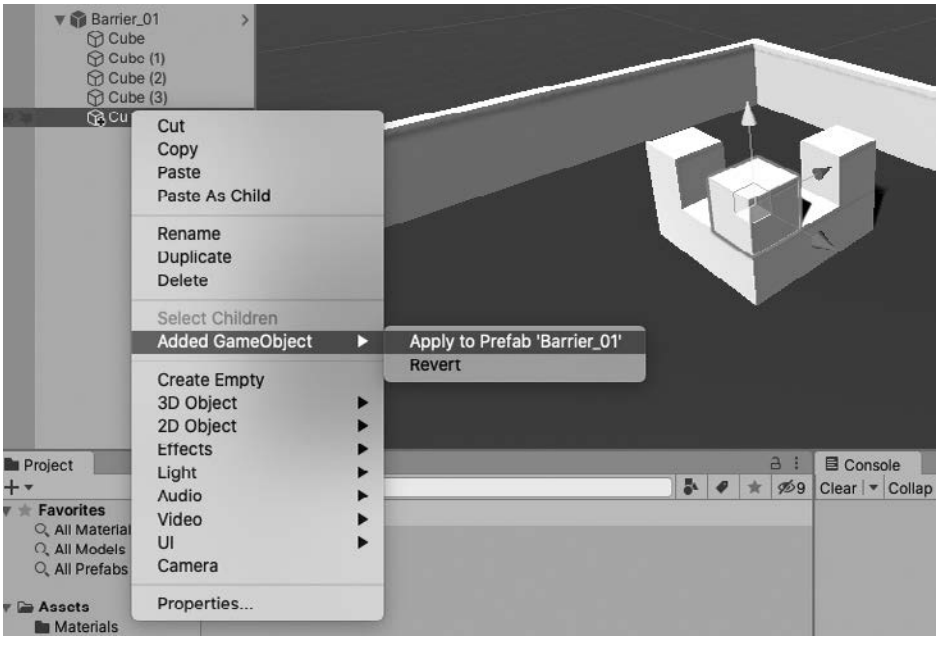


Рис. 6.17

Время действовать. Завершаем уровень

Теперь, когда у нас есть многоразовый префаб башни, построим остальную часть уровня, чтобы она соответствовала приведенному в начале раздела эскизу.

1. Скопируйте префаб `Barrier_01` три раза и поместите новые копии в разные уголки арены.
2. Создайте новый пустой `GameObject` и назовите его `Raised_Platform`.
3. Создайте куб и сформируйте из него платформу.
4. Создайте плоскость и превратите ее в пандус. Затем поверните и поместите его так, чтобы он соединял платформу с землей.
5. Скопируйте пандус, используя комбинацию клавиш `⌘+D` в Mac или `Ctrl+D` в Windows. Разместите копии пандуса.
6. Повторите предыдущий шаг еще два раза, пока у вас не появятся четыре пандуса (рис. 6.18).

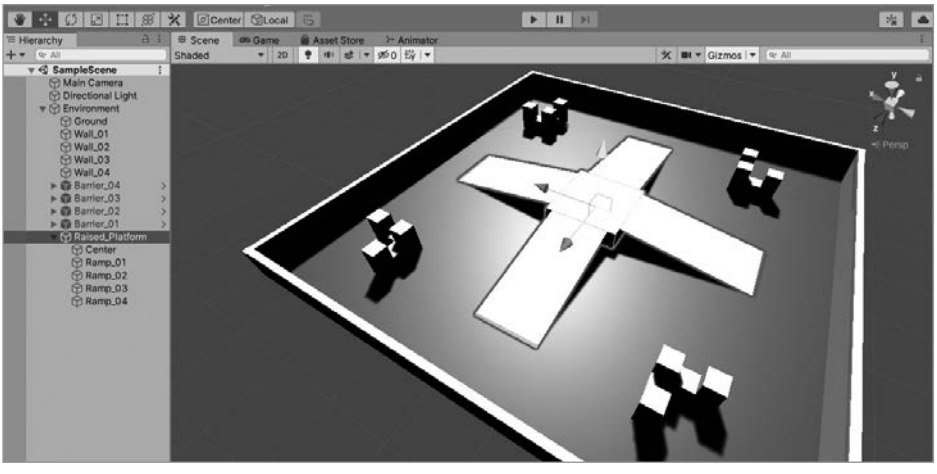


Рис. 6.18

Вы успешно набросали свой первый игровой уровень! Но пока не слишком гордитесь — мы только начинаем. Во всех хороших играх есть предметы, которые игроки могут подбирать или с которыми могут взаимодействовать. В следующем задании ваша задача — создать бонус здоровья и сделать его префабом.

Испытание героя. Создаем бонус здоровья

Разместить все эти объекты на сцене — дело не пары секунд, но оно того стоит. Теперь создадим бонус следующим образом.

1. Создайте, расположите и масштабируйте `GameObject` типа `Capsule` и назовите его `Health_Pickup`.
2. Создайте и прикрепите новый `Material` желтого цвета к объекту `Health_Pickup`.
3. Перетащите объект `Health_Pickup` в папку Prefab.

На рис. 6.19 показан пример того, как должен выглядеть готовый результат.

На этом мы завершаем нашу работу над дизайном уровня и верстkой. Далее мы пройдем ускоренный курс по освещению в Unity, а позже в данной главе узнаем, как работать с анимацией элементов.

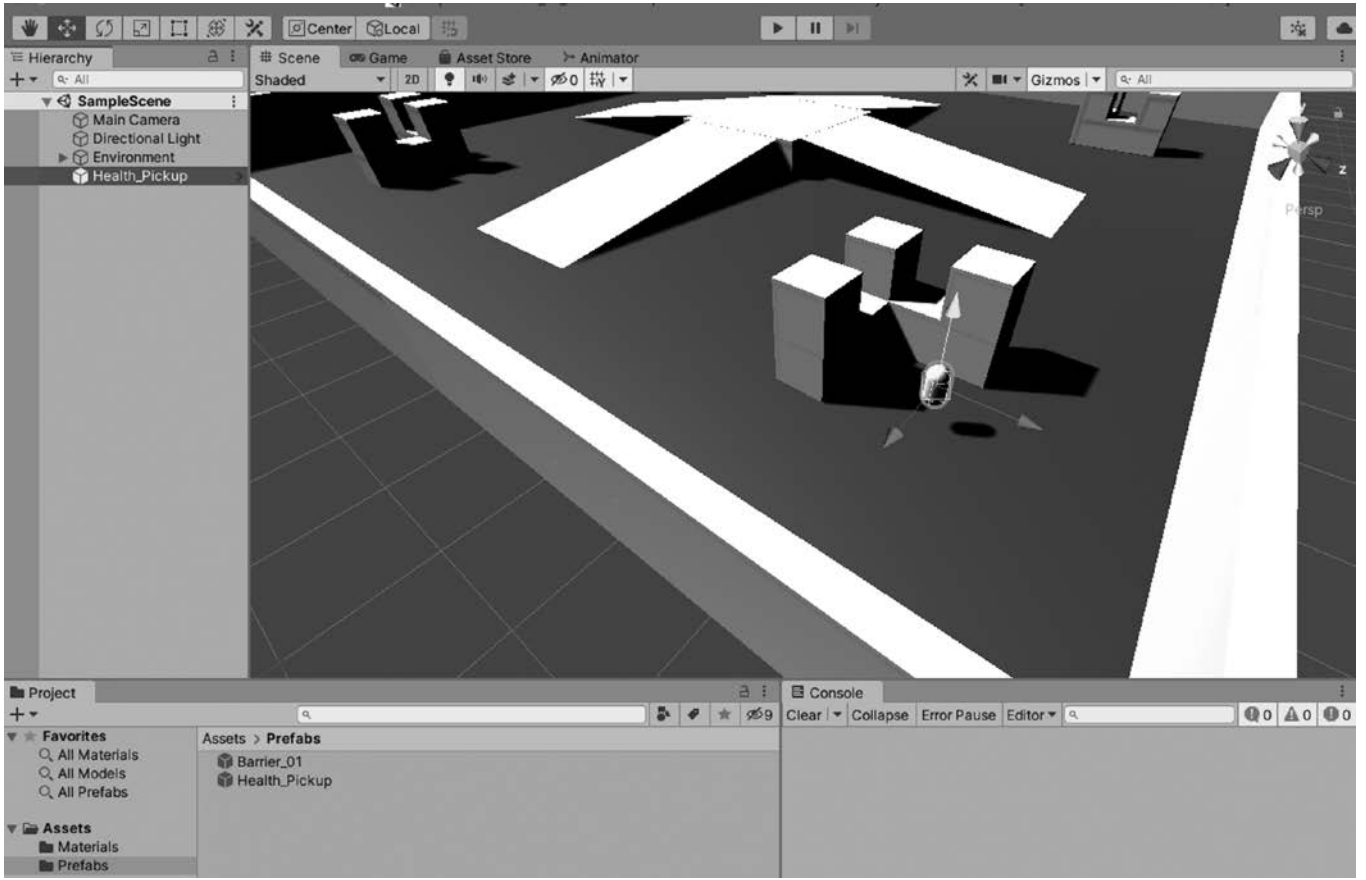

Center
Local
Hierarchy
All
SampleScene
Main Camera
Directional Light
Environment
Health_Pickup
Scene
Game
Asset Store
Animator
Shaded
2D
Gizmos
All
Persp
Project
Favorites
All Materials
All Models
All Prefabs
Assets
Materials
Prefabs
Assets > Prefabs
Barrier_01
Health_Pickup
Console
Clear
Collapse
Error Pause
Editor

Рис. 6.19

Основы работы со светом

Освещение в Unity — огромная тема, но ее можно разделить на две категории: освещение в реальном времени и предварительно вычисленное освещение. Оба типа источников света позволяют настраивать цвет и интенсивность света, а также направление, в котором он светит в сцене. Разница лишь в том, как движок Unity обсчитывает его.

Освещение в реальном времени вычисляется в каждом кадре, а это значит, что любой объект, через который свет проходит на своем пути, будет отбрасывать реалистичные тени, и в целом источник света станет вести себя как реальный. Однако это может значительно замедлить вашу игру и потребовать огромной вычислительной мощности, в зависимости от количества источников света в вашей сцене. С другой стороны, *предварительно вычисленное* освещение сохраняется в текстуре, называемой **картой освещения**, которая затем применяется к сцене, или *запекается*. Это экономит вычислительную мощность, но запеченное освещение получается статичным. То есть оно не реагирует на изменения и не меняется при перемещении объектов в сцене.

Существует также смешанный тип освещения, называемый глобальным предварительно вычисленным освещением в реальном времени, в котором стерта разница между процессами реального времени и предварительно вычисленными процессами. Это уже довольно сложная часть Unity, и потому мы не будем рассматривать ее здесь, но вы можете посмотреть документацию, пройдя по ссылке docs.unity3d.com/ru/current/Manual/LightingOverview.html.

Посмотрим, как создавать световые объекты в самой сцене Unity.

Создание источников света

По умолчанию в сцене располагается компонент `Directional Light` (рис. 6.20), который действует как основной источник освещения, но вы можете добавлять и свои источники, как и любой другой `GameObject`. Идея управления источниками света может быть для вас новой, но они являются такими же объектами в Unity, а это значит, что их можно располагать, масштабировать и вращать, как вам нужно.

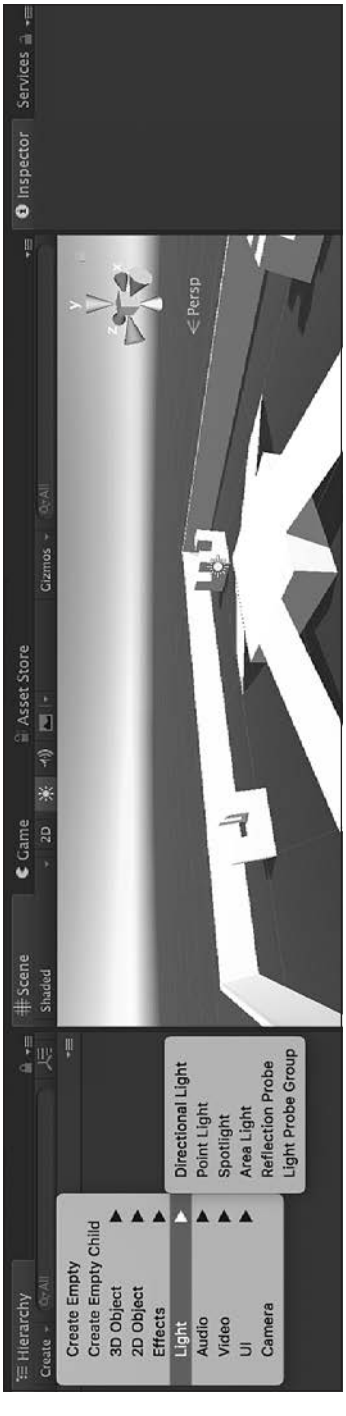

Hierarchy
Create
Create Empty
Create Empty Child
3D Object
2D Object
Effects
Light
Audio
Video
UI
Camera
Directional Light
Point Light
Spotlight
Area Light
Reflection Probe
Light Probe Group
Scene
Shaded
Game
2D
Asset Store
Gizmos
Persp
Inspector
Services

Рис. 6.20

Рассмотрим несколько примеров световых объектов в реальном времени и их производительность.

- Направленные (Directional) источники света отлично подходят для имитации естественного света, например солнечного. У них нет фактического положения в сцене, и их свет падает на все объекты в одном и том же направлении.
- Точечные (Point) источники света — это, по сути, летающие шарики, испускающие световые лучи из центральной точки во всех направлениях. Можно настраивать их положение в сцене и интенсивность света.
- Прожекторы (Spotlight) излучают свет в заданном направлении, но освещают лишь ограниченный сектор. Очень похоже на прожектора в реальном мире.

Типы Reflection Probes и Light Probe Groups в проекте Hero Born не понадобятся, однако если вам интересно, то можете почитать о них, пройдя по ссылкам docs.unity3d.com/ru/current/Manual/ReflectionProbes.html и docs.unity3d.com/ru/current/Manual/LightProbes.html.

Как и все `GameObject` в Unity, у источников света есть свойства, которые можно настраивать, чтобы придать сцене определенную атмосферу или тему.

Свойства источников света

На рис. 6.21 показан компонент `Light` в направленном источнике в нашей сцене. Все эти свойства можно настраивать для создания иммерсивных сред, но основные из них, о которых нам нужно знать, — это цвет, режим и интенсивность. Они определяют оттенок света, вычисление в реальном времени или запекание, а также общую силу света.

Как и в случае с другими компонентами Unity, к этим свойствам можно получить доступ через сценарии и класс Light, который можно найти по адресу docs.unity3d.com/ScriptReference/Light.html.

Рис. 6.21

Теперь, когда мы знаем немного больше о том, как создается освещение игровой сцены, поговорим об анимации!

Анимация в Unity

Анимация объектов в Unity бывает разной — от простого эффекта вращения до сложных движений и действий персонажей. Вся анимация настраивается в окнах Animation и Animator:

- в окне Animation создаются сегменты анимации, называемые клипами, и управление ими осуществляется с помощью временной шкалы. Свойства объекта записываются на этой временной шкале, а затем воспроизводятся для создания анимационного эффекта;
- окно Animator управляет этими клипами и их переходами с помощью объектов, называемых контроллерами анимации.

Более подробную информацию об окне Animator и его контроллерах можно получить на странице docs.unity3d.com/ru/current/Manual/AnimatorControllers.html.

Создание клипов

Любой `GameObject`, к которому вы хотите применить анимационный клип, должен иметь компонент `Animator` с набором `Animation Controller`. Если при создании нового клипа в проекте нет контроллера, то Unity создаст его и сохранит в месте расположения клипов, и контроллер затем можно использовать для управления своими клипами. Ваша следующая задача — создать анимацию для подбираемого бонуса.

Время действовать. Создаем новый клип

Мы собираемся анимировать префаб `Health_Pickup` простой анимацией бесконечного вращения. Чтобы создать новый клип, необходимо выполнить следующие действия.

1. Выберите команду Window ▸ Animation ▸ Animation, чтобы открыть панель Animation, и закрепите ее рядом с панелью Console.
2. Выберите на панели Hierarchy префаб `Health_Pickup`, а затем нажмите кнопку Create на панели Animation (рис. 6.22).
3. Создайте новую папку из следующего раскрывающегося списка, назовите ее Animations, а затем назовите новый клип `Pickup_Spin` (рис. 6.23).
4. Убедитесь, что на панели Animation появился новый клип (рис. 6.24).

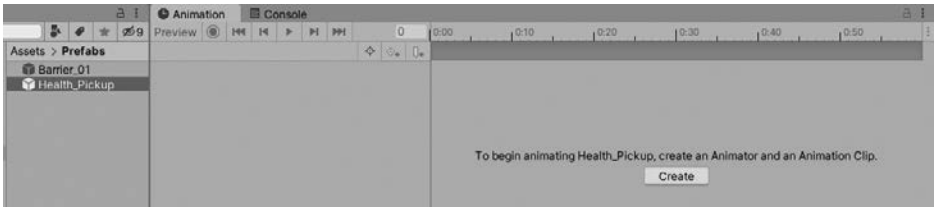


Рис. 6.22

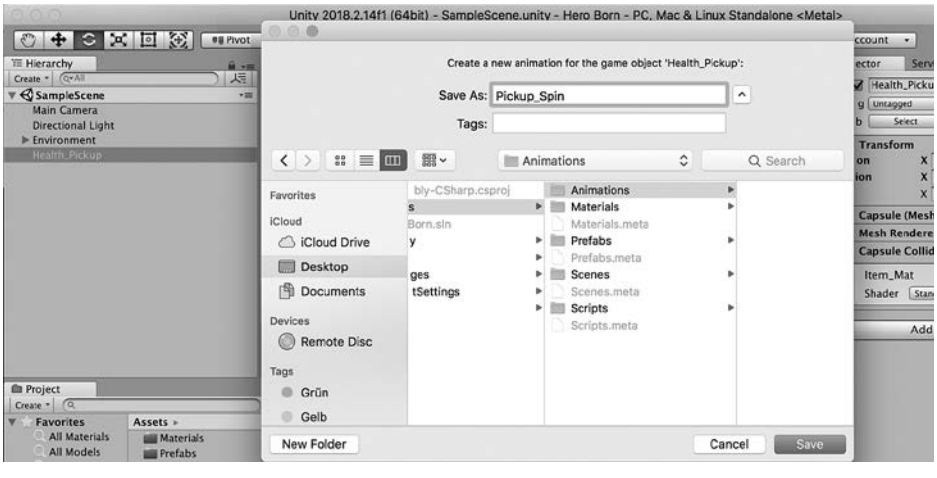


Рис. 6.23

Рис. 6.24

Поскольку у нас не было контроллеров Animator, Unity сам создал контроллер под названием `Health_Pickup` в папке Animation. Выбрав `Health_Pickup`

при создании клипа, мы также добавили в префаб компонент `Animator` с контроллером `Health_Pickup` (рис. 6.25).

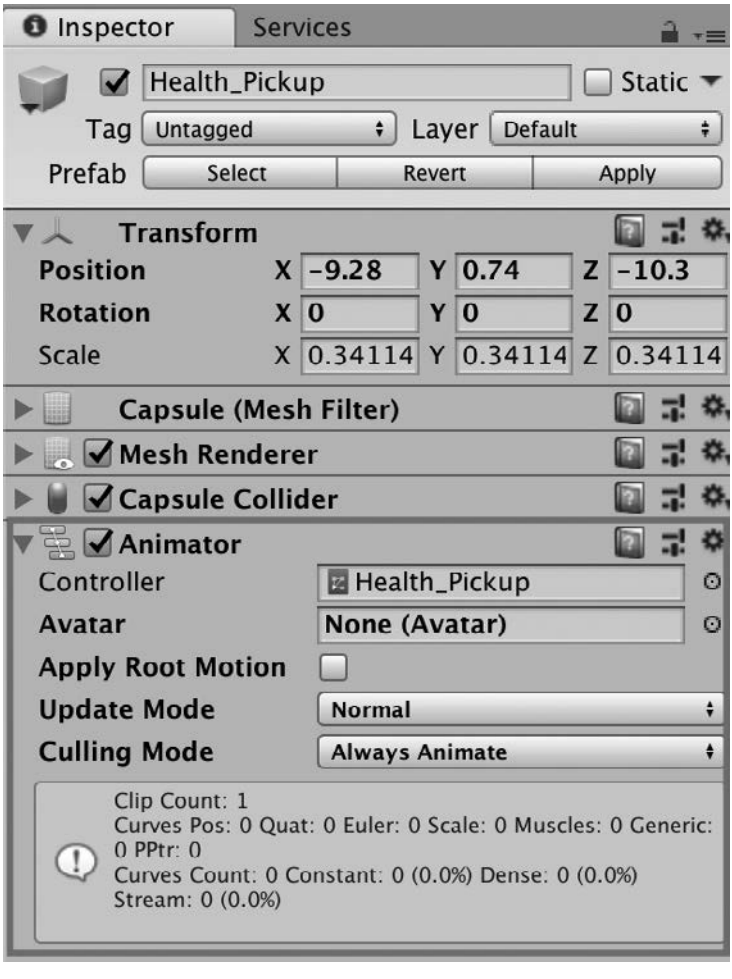


Рис. 6.25

В клипах с движением, как и в киносъемке, мы работаем с кадрами. По мере перемещения клипа по кадрам анимация воспроизводится, создавая эффект движения. Пока в Unity ничего не изменилось. Нам нужно записать наш целевой объект в различных положениях в разных кадрах, чтобы Unity мог воспроизвести клип.

Запись ключевых кадров

Теперь, когда у нас есть клип, с которым можно работать, вы увидите на панели Animation пустую временную шкалу. По сути, когда мы изменяем поворот по оси `z` префаба `Health_Pickup` или любое другое свойство, которое можно анимировать, шкала времени будет записывать эти изменения как ключевые кадры. Затем Unity объединяет их в полную анимацию, подобно тому как отдельные кадры аналогового фильма превращаются в движущееся изображение.

Взгляните на рис. 6.26 и запомните расположение кнопки записи и временной шкалы.

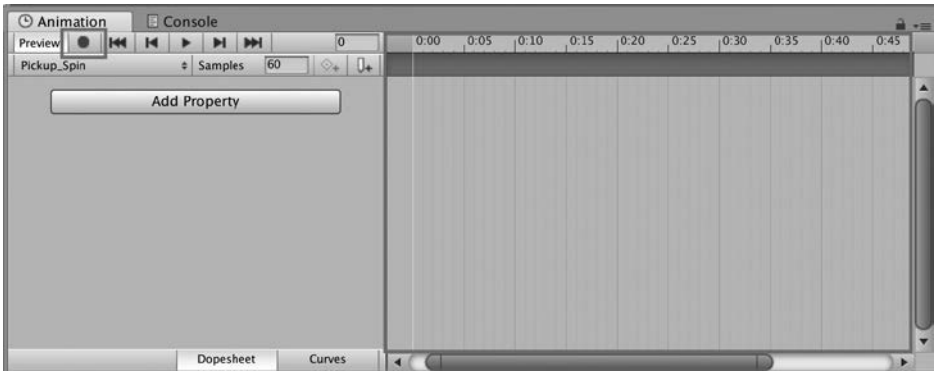


Рис. 6.26

Теперь покрутим наш бонус.

Время действовать. Вращаем анимацию

Мы хотим, чтобы префаб `Health_Pickup` совершал полный поворот на 360 градусов по оси `z` каждую секунду. Для этого достаточно задать три ключевых кадра и позволить Unity сделать остальное самому.

1. Выберите команду Add Property ▸ Transform и нажмите знак + рядом со свойством Rotation (рис. 6.27).
2. Нажмите кнопку Record, чтобы начать анимацию.

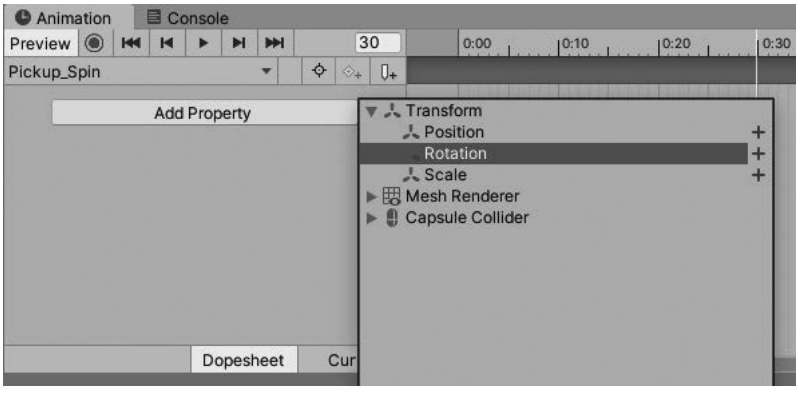


Рис. 6.27

3. Поместите курсор на отметку `0:00` на шкале времени, а значение поворота `Health_Pickup` по оси `z` оставьте равным `0`:
   а) поместите курсор на отметку `0:30` на шкале времени и установите угол поворота равным `180`;
   б) поместите курсор на `1:00` на шкале времени и установите поворот по оси `z` равным `360` (рис. 6.28).

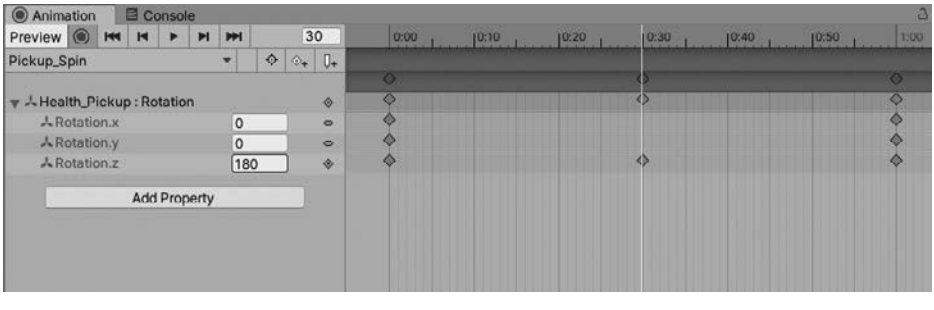


Рис. 6.28

4. Нажмите кнопку Record, чтобы завершить анимацию.
5. Нажмите кнопку Play справа от кнопки записи, чтобы просмотреть цикл анимации.

Теперь объект `Health_Pickup` будет вращаться по оси `z`, сменяя значения `0`, `180` и `360` градусов каждую секунду, и в результате мы получили

циклическую анимацию вращения. Если вы сейчас запустите игру, то анимация станет работать бесконечно, пока игра не будет остановлена (рис. 6.29).

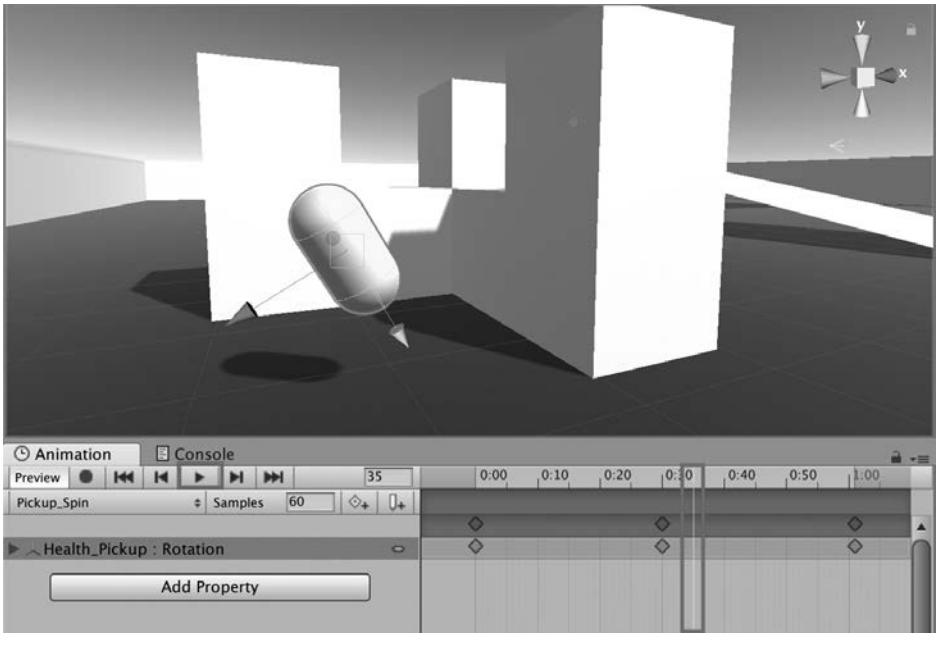


Рис. 6.29

Все анимации имеют кривые, которые более подробно определяют, как выполняется анимация. Мы не станем углубляться в эту тему, но основы понимать важно. Мы рассмотрим их в следующем подразделе.

Кривые и касательные

Помимо анимации свойства объекта, Unity позволяет нам управлять воспроизведением анимации во времени с помощью кривых анимации. До этого мы работали в режиме Dopesheet, переключение на который выполняется в нижней части окна анимации. Если вы щелкнете на вкладке Curves (показана на рис. 6.30), то увидите другой график с акцентными точками вместо записанных ключевых кадров. Мы хотим,

чтобы анимация вращения была плавной (линейной), поэтому оставим все как есть. Однако вы можете подрегулировать ускорение, замедление или изменение анимации в любой момент с помощью точек.

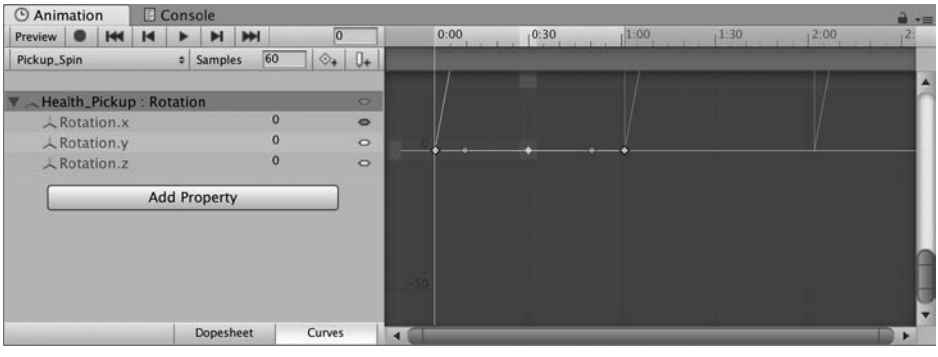


Рис. 6.30

Поскольку кривые анимации определяют изменение свойств во времени, нам необходимо устранить «скачок», который возникает каждый раз, когда анимация `Health_Pickup` повторяется. Для этого нам нужно изменить свойство `Tangent` анимации, которое определяет, как ключевые кадры переходят один в другой. Доступ к этому параметру можно получить, щелкнув правой кнопкой мыши на любом ключевом кадре на временной шкале в режиме Dopesheet, как показано ниже (рис. 6.31).

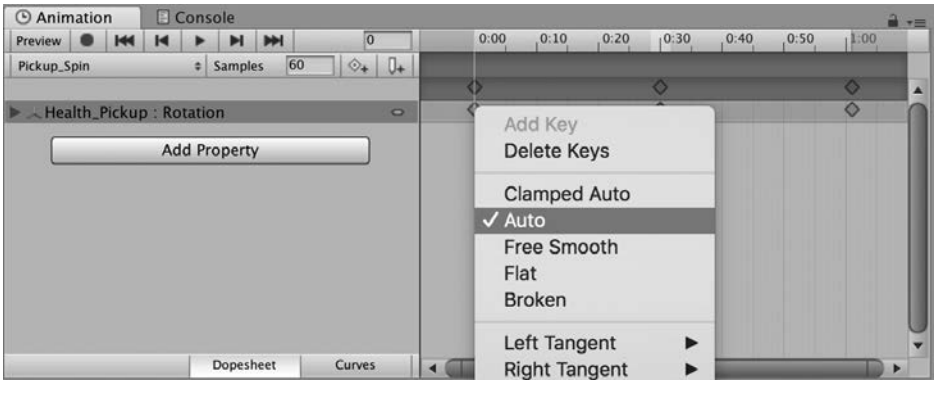


Рис. 6.31

Обе эти темы по сложности уже не относятся к начальному уровню, поэтому мы не будем углубляться в них. Если вам интересно, то можете взглянуть на документацию по кривым анимации и параметрам касательной, пройдя по ссылке docs.unity3d.com/ru/current/Manual/animeditor-AnimationCurves.html.

Если мы запустим вращение сейчас, то увидим небольшую паузу между циклами вращения нашего бонуса. Ваша задача — сгладить эту паузу, чем мы и займемся дальше.

Время действовать. Сглаживаем вращение

Настроим касательные на первом и последнем кадрах анимации, чтобы вращающаяся анимация плавно переходила в следующий цикл.

1. Щелкните правой кнопкой мыши на ромбиках первого и последнего ключевых кадров на временной шкале и выберите параметр Auto (рис. 6.32).

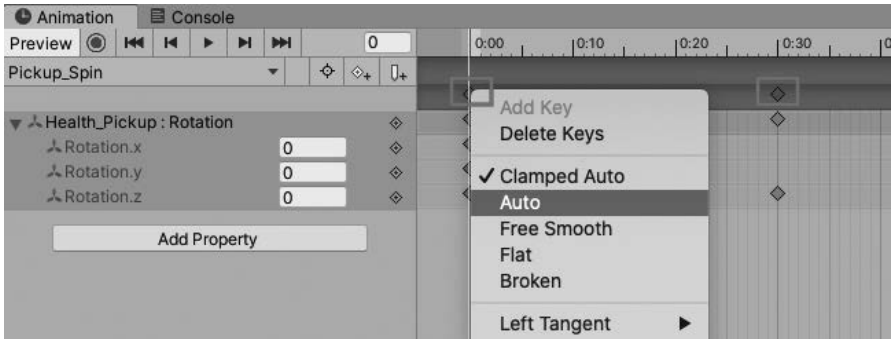


Рис. 6.32

2. Переместите объект `Main Camera` так, чтобы при запуске игры был виден объект `Health_Pickup`, и нажмите кнопку Play (рис. 6.33).

Изменение параметра касательных первого и последнего ключевых кадров на Auto дает Unity указание сделать переходы плавными, и за счет этого рывки при зацикливании анимации исчезнут.

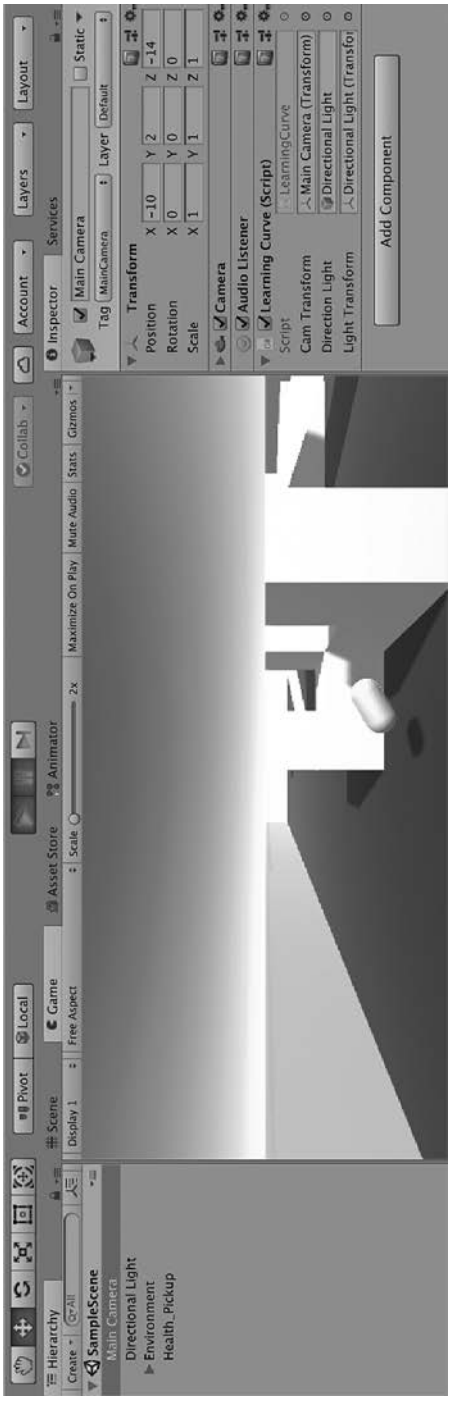

Hierarchy
Create
SampleScene
Main Camera
Directional Light
Environment
Health_Pickup
Scene
Game
Asset Store
Animator
Pivot
Local
Collab
Account
Layers
Layout
Inspector
Services
Main Camera
Static
Tag
MainCamera
Layer
Default
Transform
Position
Rotation
Scale
Camera
Audio Listener
Learning Curve (Script)
Script
LearningCurve
Cam Transform
Main Camera (Transform)
Direction Light
Directional Light
Light Transform
Directional Light (Transfo
Add Component

Рис. 6.33

Объекты можно анимировать и с помощью C#, управляя определенными свойствами, такими как position или rotation. Несмотря на то что мы не будем углубляться здесь в эту конкретную тему, вам важно знать, что в Unity можно программировать анимацию.

Это все, что понадобится для данной книги с точки зрения анимации, но я бы посоветовал вам ознакомиться со всем набором инструментов Unity в этой области. Ваши игры станут более увлекательными и ваши игроки будут вам благодарны! Далее мы кратко поговорим о системе частиц Unity и о том, как добавить в сцену эффекты.

Система частиц

Когда дело доходит до создания эффектов движения, таких как взрывы или реактивные струи космического корабля, вам пригодятся эффекты частиц Unity.

Системы частиц испускают спрайты или сетки, которые мы называем частицами, совместно создающие цельный эффект. Свойства частиц, от их цвета и размера до времени их «жизни» и скорости перемещения в заданном направлении, можно настраивать. Их можно создавать как отдельные объекты или объединять в целях создания более реалистичных эффектов.

Эффекты системы частиц могут быть чрезвычайно сложными, и их можно использовать для создания практически всего, что вы можете себе представить. Однако умение создавать реалистичные эффекты требует практики. Для начала ознакомьтесь с руководством по частицам: docs.unity3d.com/ru/current/Manual/Built-inParticleSystem.html.

Наш бонус все еще выглядит немного блеклым, хотя и крутится. Добавим ему визуальный эффект.

Время действовать. Добавляем эффект искр

Нам нужно привлечь внимание игрока к собираемым предметам, которые мы разместим по уровню, поэтому добавим объекту `Health_Pickup` простой эффект частиц.

1. Выберите команду Create ▸ Effects ▸ Particle System на панели Hierarchy (рис. 6.34).

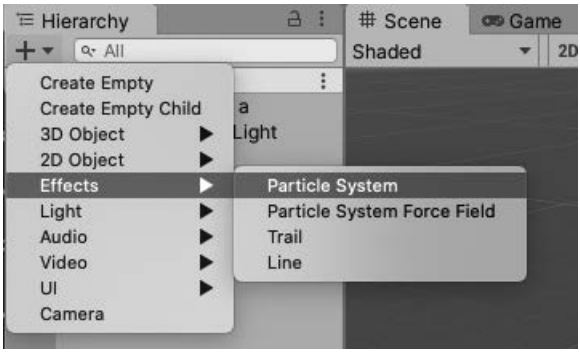


Рис. 6.34

2. Разместите объект `Particle System` в середине объекта `Health_Pickup`.
3. Выберите новую систему частиц и обновите следующие свойства на панели Inspector:
   - Start Lifetime: `2`;
   - Start Speed: `-0.25`;
   - Start Size: `0.75`;
   - Start Color — оранжевый или другой цвет по вашему выбору.
4. Откройте вкладку Emissions и задайте параметр Rate Over Time равным `5`.
5. Откройте вкладку Shape и задайте параметр Shape равным Sphere (рис. 6.35).

Созданный нами объект `Particle System` теперь будет рендерить и испускать частицы в каждом кадре, основываясь на свойствах, которые мы установили на панели Inspector.

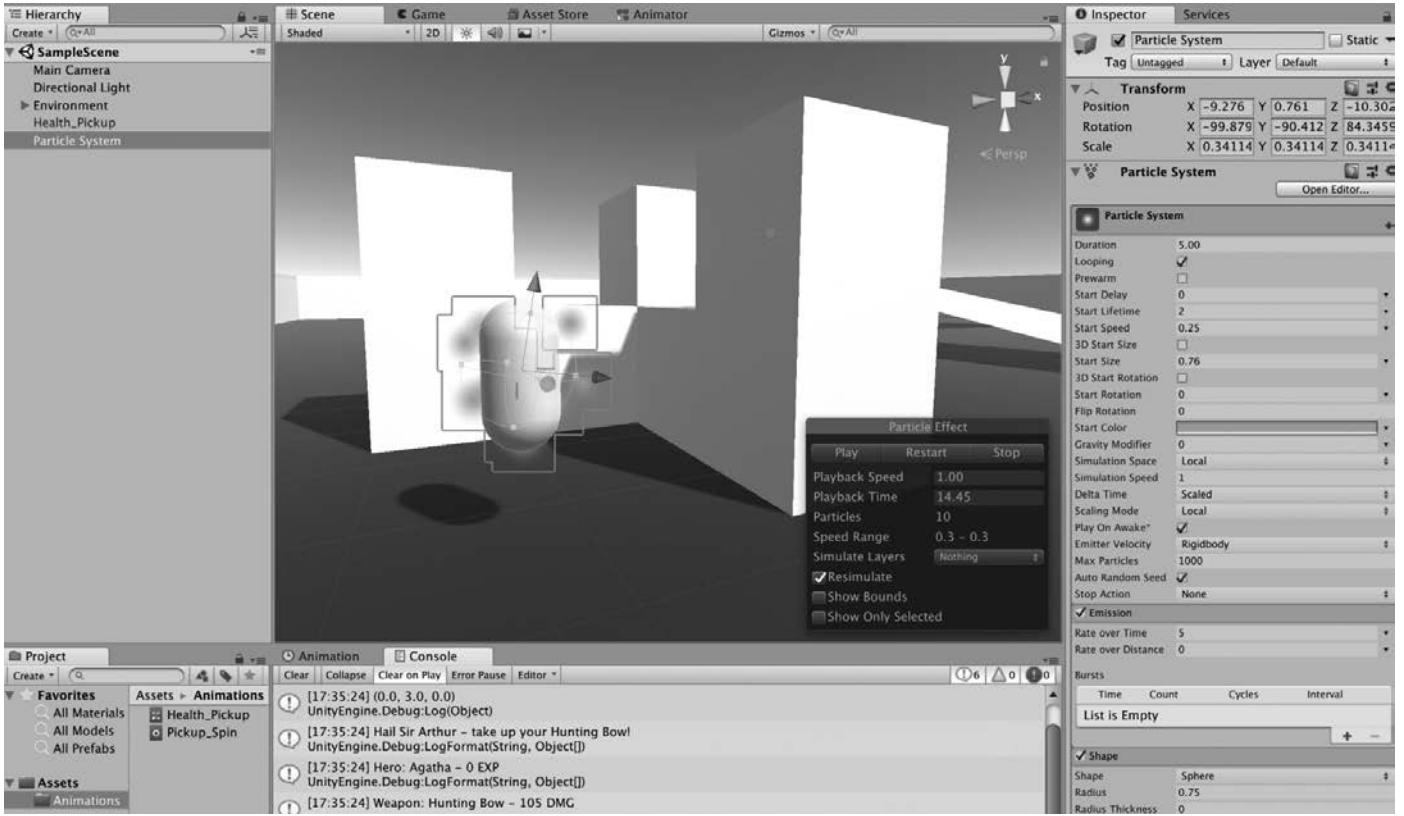

Hierarchy
Create
SampleScene
Main Camera
Directional Light
Environment
Health_Pickup
Particle System
Scene
Game
Asset Store
Animator
Shaded
2D
Gizmos
Persp
Particle Effect
Play
Restart
Stop
Playback Speed 1.00
Playback Time 14.45
Particles 10
Speed Range 0.3 - 0.3
Simulate Layers Nothing
Resimulate
Show Bounds
Show Only Selected
Inspector
Services
Particle System
Static
Tag Untagged
Layer Default
Transform
Position X -9.276 Y 0.761 Z -10.302
Rotation X -99.879 Y -90.412 Z 84.3459
Scale X 0.34114 Y 0.34114 Z 0.3411
Particle System
Open Editor...
Particle System
Duration 5.00
Looping
Prewarm
Start Delay 0
Start Lifetime 2
Start Speed 0.25
3D Start Size
Start Size 0.76
3D Start Rotation
Start Rotation 0
Flip Rotation 0
Start Color
Gravity Modifier 0
Simulation Space Local
Simulation Speed 1
Delta Time Scaled
Scaling Mode Local
Play On Awake*
Emitter Velocity Rigidbody
Max Particles 1000
Auto Random Seed
Stop Action None
Emission
Rate over Time 5
Rate over Distance 0
Bursts
Time Count Cycles Interval
List is Empty
Shape
Shape Sphere
Radius 0.75
Radius Thickness 0
Project
Create
Favorites
All Materials
All Models
All Prefabs
Assets
Animations
Assets ▸ Animations
Health_Pickup
Pickup_Spin
Animation
Console
Clear
Collapse
Clear on Play
Error Pause
Editor
[17:35:24] (0.0, 3.0, 0.0)
UnityEngine.Debug:Log(Object)
[17:35:24] Hail Sir Arthur – take up your Hunting Bow!
UnityEngine.Debug:LogFormat(String, Object[])
[17:35:24] Hero: Agatha – 0 EXP
UnityEngine.Debug:LogFormat(String, Object[])
[17:35:24] Weapon: Hunting Bow – 105 DMG

Рис. 6.35

Подведем итоги

Вы изучили еще одну главу, в которой было много нового, особенно для тех из вас, кто плохо знаком с Unity. Несмотря на то что эта книга в большей степени посвящена языку C#, мы вынуждены выделить время, чтобы рассмотреть разработку игр, документацию и функции движка, не связанные с написанием сценариев. Хотя у нас не было времени на подробное рассмотрение инструментов освещения, анимации и системы частиц, вам стоит познакомиться с ними, если вы захотите работать в Unity дальше.

В следующей главе мы вернемся к программированию основной механики игры Hero Born, начав с настройки подвижного объекта игрока, управления камерой и понимания того, как физическая система Unity управляет игровым миром.

Контрольные вопросы. Основные функции Unity

1. К какому типу `GameObject` относятся кубы, капсулы и сферы?
2. Какую ось Unity использует для представления глубины, которая придает сцене трехмерный вид?
3. Как превратить `GameObject` в многоразовый префаб?
4. Какие единицы измерения используются в системе анимации Unity для записи анимации объектов?

7 Движение, управление камерой и столкновения

Первое, что делает игрок при запуске новой игры, — знакомится с управлением персонажем и камерой. Это всегда интересное занятие, которое позволяет игроку понять, какой игровой процесс ждет его дальше. Персонаж в игре Hero Born будет представлять собой капсулу, которую можно будет перемещать и вращать с помощью клавиш `W`, `A`, `S`, `D` или клавиш со стрелками.

Сначала изучим, как можно управлять компонентом объекта `Transform`, а затем воспроизведем ту же схему управления, но через приложение силы. Это позволит получить более реалистичный эффект движения. Когда мы перемещаем игрока, камера будет следовать за ним, находясь чуть позади и выше, что упростит нам прицеливание при реализации механики стрельбы. Наконец, мы исследуем, как физический движок Unity обрабатывает столкновения и физические взаимодействия. В этом нам поможет уже созданный префаб бонуса.

Затем мы объединим все это на игровом уровне, но пока без какой-либо механики действия. Мы получим первое представление о том, как язык C# используется для программирования игровых функций, изучив такие темы, как:

- реализация движения и вращения;
- обработка ввода со стороны игрока;
- программирование поведения камеры;
- физика в Unity и приложенная сила;
- базовые коллайдеры и обнаружение столкновений.

Перемещение игрока

Размышляя о том, как лучше всего перемещать вашего персонажа по виртуальному миру, следует выбрать вариант, который выглядит наиболее реалистично, а не загружать игру дорогостоящими вычислениями.

В большинстве случаев приходится находить какой-то компромисс, и Unity не исключение.

Существует три наиболее распространенных способа перемещения `GameObject`.

- **Вариант 1:** использовать компонент `Transform` объекта `GameObject` для реализации перемещения и вращения. Это самое простое решение, и именно с него мы начнем.
- **Вариант 2:** прикрепить к `GameObject` компонент `Rigidbody` и запрограммировать применение силы. Это решение реализуется за счет физической системы Unity, которая за нас будет делать всю тяжелую работу, создавая более реалистичный эффект движения. Позже в главе мы перепишем наш код и реализуем данный подход, опробовав таким образом оба метода.

Unity предлагает придерживаться последовательного подхода при перемещении или вращении GameObject: пользоваться либо компонентом Transform, либо Rigidbody, но не обоими одновременно.

- **Вариант 3:** взять готовый компонент или префаб Unity, например `Character Controller` или `FirstPersonController`. В этом случае мы сократим объем шаблонного кода и получим реалистичный эффект, сокращая время создания прототипа.

Вы можете найти больше информации о компоненте Character Controller и его использовании, пройдя по ссылке docs.unity3d.com/ScriptReference/CharacterController.html.

Префаб FirstPersonController есть в пакете Standard Assets, который можно скачать отсюда: assetstore.unity.com/packages/essentials/asset-packs/standard-assets-32351.

Поскольку мы только начинаем работать с движением игрока в Unity, начнем мы с прямой работы с компонентом `Transform`, а позже в этой главе перейдем к физике `Rigidbody`.

Создание игрока

Для игры Hero Born выберем вид на игрока от третьего лица. Сначала создим капсулу, которой можно будет управлять с помощью клавиатуры, и камеру, которая станет следить за капсулой во время ее движения. Несмотря на то что эти два `GameObject` будут всегда вместе, их сценарии разделим на два файла, чтобы упростить задачу контроля.

Прежде чем можно будет выполнить какой бы то ни было сценарий, нужно сначала добавить в сцену капсулу игрока. Этим и займемся.

Время действовать. Создаем капсулу игрока

Создать красивую капсулу игрока будет просто, как «раз-два-три».

1. Выберите команду Create ▸ 3D Object ▸ Capsule на панели Hierarchy и назовите созданную капсулу `Player`.
2. Выберите объект `Player` и нажмите кнопку Add Component в нижней части панели Inspector. Найдите компонент `Rigidbody` и нажмите клавишу Enter, чтобы добавить его.
3. Разверните свойство `Constraints` в нижней части компонента `Rigidbody`:
   - установите флажки свойства `Freeze Rotation` по осям *X* и *Y*.
4. Выберите папку Materials и нажмите Create ▸ Material. Назовите новый материал `Player_Mat`.
5. Измените свойство `Albedo` этого материала на ярко-зеленый и перетащите материал на объект `Player` на панели Hierarchy (рис. 7.1).


Hierarchy
SampleScene
Main Camera
Directional Light
Environment
Health_Pickup
Particle System
Player
Scene
Game
Asset Store
Animator
Shaded
2D
Gizmos
Project
Favorites
All Materials
All Models
All Prefabs
Assets
Animations
Materials
Prefabs
Scenes
Scripts
Packages
Ground_Mat
Pickup_Mat
Player_Mat
Animation
Console
Clear
Collapse
Error Pause
Editor
Inspector
Player
Static
Tag
Untagged
Layer
Default
Transform
Position
Rotation
Scale
Capsule (Mesh Filter)
Mesh Renderer
Capsule Collider
Rigidbody
Mass
Drag
Angular Drag
Use Gravity
Is Kinematic
Interpolate
Collision Detection
Discrete
Constraints
Info
Player_Mat
Shader
Standard
Add Component
Account
Layers
Layout

Рис. 7.1

Вы создали объект `Player` из примитива капсулы, компонента `Rigidbody` и нового ярко-зеленого материала. Пока не нужно задумываться о том, что представляет собой компонент `Rigidbody`. Сейчас нам достаточно знать, что он позволяет нашей капсуле взаимодействовать с физической системой. Об этом мы поговорим более подробно в конце главы, когда будем рассматривать принципы работы физической системы Unity. Ну а пока нам нужно поговорить об очень важной теме в трехмерном пространстве — о векторах.

Введение в векторы

Теперь, когда капсула игрока и камера настроены, мы можем начать разговор о том, как перемещать и вращать `GameObject`, используя его компонент `Transform`. Методы `Translate` и `Rotate` являются частью встроенного в Unity класса `Transform`, и для выполнения преобразования им нужен векторный параметр.

В Unity векторы служат для хранения данных о положении и направлении в 2D- и 3D-пространствах, и поэтому они бывают двух разновидностей — `Vector2` и `Vector3`. Они используются так же, как и любой другой тип переменных, с которыми мы уже работали, но хранят другую информацию. Поскольку наша игра трехмерная, мы задействуем объекты `Vector3`, поэтому для их создания будут нужны значения `x`, `y` и `z`. Для 2D-векторов требуются только значения `x` и `y`. Помните, что ваше положение в 3D-сцене отображается с помощью замысловатой фигурки в правом верхнем углу экрана (рис. 7.2).

Если вы хотите больше узнать о векторах в Unity, то обратитесь к документации и Scripting Reference: docs.unity3d.com/ScriptReference/Vector3.html.

Например, если мы хотим создать новый вектор для хранения исходной позиции нашей сцены, то можно использовать следующий код:

```
Vector3 origin = new Vector(0f, 0f, 0f);
```

Здесь все просто: мы создали новую переменную `Vector3`, а затем присвоили ей значения `0` для позиции `x`, `0` для позиции `y` и `0` для позиции `z`

в правильном порядке. Значения с плавающей запятой можно записывать как с точкой, так и без нее, но в конце всегда нужно добавлять букву `f`.

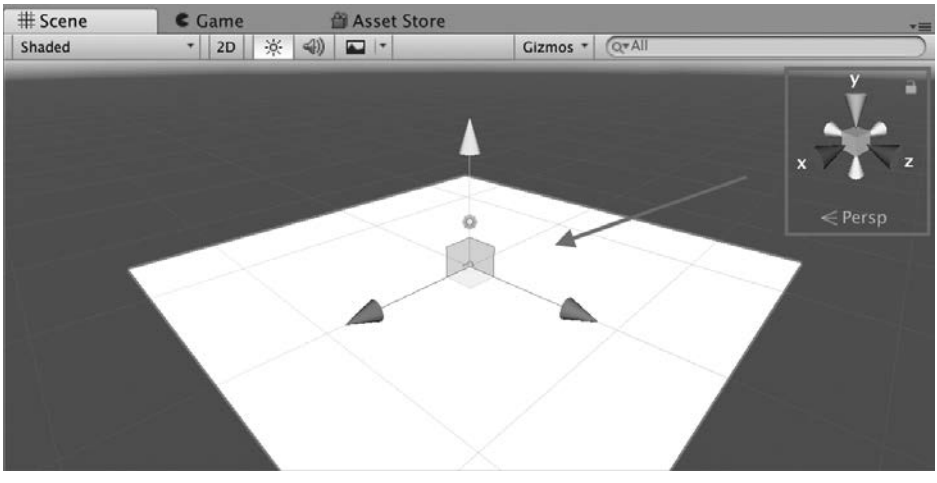


Рис. 7.2

Мы также можем создавать направленные векторы, используя свойства классов `Vector2` и `Vector3`:

```
Vector3 forwardDirection = Vector3.forward;
```

Атрибут `forwardDirection` хранит не данные о положении, а ссылается на прямое направление в нашей сцене вдоль оси *z* в трехмерном пространстве. Мы рассмотрим использование векторов позже в этой главе, а пока просто необходимо привыкнуть, что в трехмерном движении всегда нужно знать положение и направление по осям *x*, *y* и *z*.

Не волнуйтесь, если концепция векторов для вас в новинку и пока непонятна. Это сложная тема. Справочник Unity по векторам поможет вам вникнуть в тему: docs.unity3d.com/Manual/VectorCookbook.html.

Теперь, когда вы немного больше понимаете в векторах, можно приступить к реализации основ перемещения капсулы игрока. Для этого предстоит обрабатывать данные, вводимые игроком с клавиатуры, о чем и поговорим в следующем подразделе.

Обработка ввода от игрока

Информация о положении и направлении полезна сама по себе, но без участия игрока реализовать движение не выйдет. И здесь на помощь приходит класс `Input`, который обрабатывает все: от нажатий клавиш и положения мыши до данных акселерометров и гироскопов.

Для перемещения в игре Hero Born будут использоваться клавиши W, A, S, D и стрелки, а также сценарий, который заставит камеру следовать туда, куда игрок указывает с помощью мыши. Для этого нам нужно понять, как работают оси ввода.

Сначала перейдите в меню Edit ▸ Project Settings ▸ Input и откройте панель Input Manager, показанную на рис. 7.3.

Перед вами появится длинный список настроек вводов в Unity по умолчанию, но мы в качестве примера возьмем ось `Horizontal`. По умолчанию на ней в качестве источника ввода для положительного и отрицательного значений установлены значения `left` и `right`, а в качестве `Alt Negative` и `Alt Positive` назначены клавиши `a` и `d`.

При каждом запросе из кода оси ввода ее значение будет находиться в диапазоне от –1 до 1. Например, когда игрок нажимает стрелку влево или кнопку `A`, горизонтальная ось регистрирует значение –1. Когда игрок отпускает клавишу, значение снова становится равным 0. Аналогично, когда используются стрелка вправо или клавиша `D`, горизонтальная ось регистрирует значение 1. В результате мы можем обрабатывать четыре разных входных значения для одной оси с помощью только одной строки кода, а не писать множество операторов `if-else` для каждого варианта нажатия.

Захват входных осей осуществляется с помощью простого вызова метода `Input.GetAxis()` с указанием имени желаемой оси. Так мы и поступим с осями `Horizontal` и `Vertical` ниже. В качестве дополнительного преимущества Unity применяет сглаживающий фильтр, который делает частоту считывания входных кадров независимой. Вы также можете использовать метод `Input.GetAxisRaw()`, у которого нет сглаживающего фильтра. Документацию по методу `GetAxisRaw` можно найти, пройдя по ссылке docs.unity3d.com/ScriptReference/Input.GetAxisRaw.html.

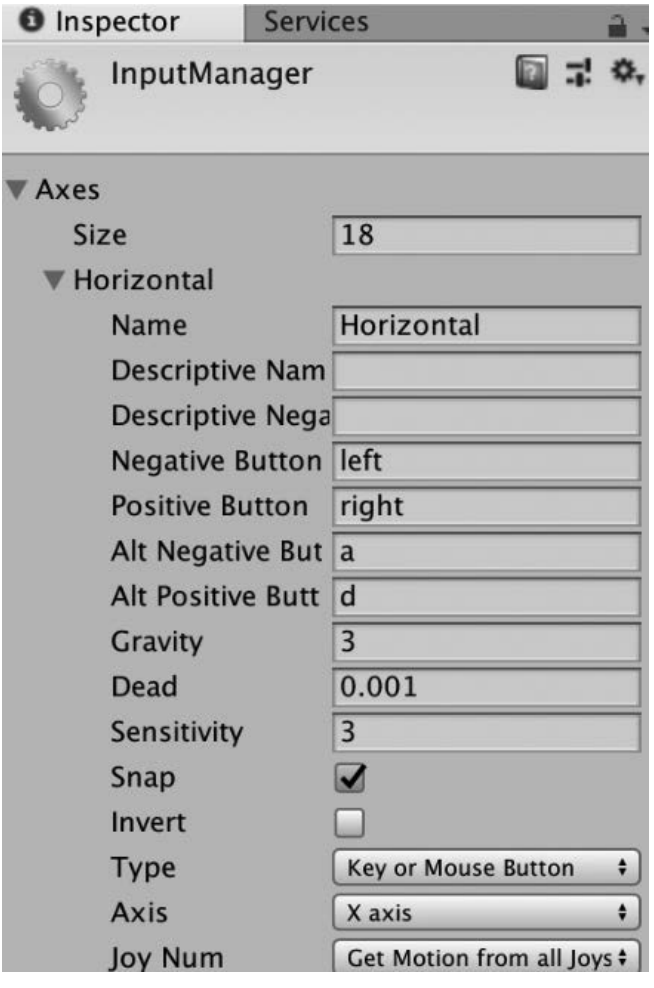


Рис. 7.3

Настройки вводов по умолчанию можно менять как угодно, а еще вы можете создавать собственные оси, увеличивая свойство Size в Input Manager на 1 и задавая имя для вновь созданной копии.

Unity недавно выпустила новую систему ввода, чтобы увеличить число поддерживаемых платформ. Мы не будем использовать ее в нашем проекте, поскольку это не требуется, но если вам интересно, то прочитайте статью: blogs.unity3d.com/2019/10/14/introducing-the-new-input-system.

Теперь научим игрока двигаться с помощью системы ввода со стороны игрока Unity, а также напишем собственный сценарий передвижения.

Время действовать. Учим игрока двигаться

Прежде чем научить игрока двигаться, вам нужно прикрепить к капсуле игрока сценарий.

1. Создайте в папке Scripts новый сценарий C#, назовите его `PlayerBehavior` и перетащите его на капсулу `Player`.
2. Добавьте в сценарий следующий код и сохраните файл:

```
public class PlayerBehavior : MonoBehaviour
{
    // 1
    public float moveSpeed = 10f;
    public float rotateSpeed = 75f;

    // 2
    private float vInput;
    private float hInput;

    void Update()
    {
        // 3
        vInput = Input.GetAxis("Vertical") * moveSpeed;

        // 4
        hInput = Input.GetAxis("Horizontal") * rotateSpeed;

        // 5
        this.transform.Translate(Vector3.forward * vInput *
            Time.deltaTime);

        // 6
        this.transform.Rotate(Vector3.up * hInput *
            Time.deltaTime);
    }
}
```

Использовать ключевое слово this не обязательно. Visual Studio 2019 может предложить вам удалить его, чтобы упростить код, но мне кажется, лучше его оставить для ясности.

Когда в коде появляются пустые методы, такие как Start в данном сценарии, их обычно удаляют, чтобы в коде был порядок. Однако если с ними вы чувствуете себя спокойнее, то это тоже нормально. Ориентируйтесь на ваши предпочтения.

Разберем этот код.

1. Объявляем две публичные переменные, которые будут использоваться как коэффициенты:
   - переменная `movespeed` определяет, насколько быстро игрок будет двигаться вперед и назад;
   - переменная `rotateSpeed` определяет, насколько быстро игрок должен вращаться влево и вправо.
2. Объявляем две приватные переменные для хранения входных данных от игрока. Начального значения у них не будет:
   - в переменной `vInput` хранится ввод вертикальной оси;
   - в переменной `hInput` хранится ввод горизонтальной оси.
3. Метод `Input.GetAxis("Vertical")` перехватывает нажатие стрелки вверх, стрелки вниз или клавиш `W` или `S` и умножает значение оси на коэффициент `moveSpeed`:
   - стрелка вверх и клавиша `W` возвращают значение `1`, что перемещает игрока вперед, в прямом (положительном) направлении;
   - стрелка вниз и клавиша `S` возвращают `-1`, что перемещает игрока назад, в отрицательном направлении.
4. Метод `Input.GetAxis("Horizontal")` перехватывает нажатие стрелки влево, стрелки вправо или клавиш `A` и `D` и умножает значение оси на коэффициент `rotateSpeed`:
   - стрелка вправо и клавиша `D` возвращают значение `1`, поворачивая капсулу вправо;
   - стрелка влево и клавиша `A` возвращают `-1`, поворачивая капсулу влево.

Если вам интересно, можно ли выполнять весь расчет движения в одной строке, то ответ — да. Однако лучше разбить код на части, даже при условии, что вы единственный, кто будет его читать.

5. Используем метод `Translate`, который принимает параметр `Vector3`, чтобы изменить компонент `Transform` капсулы:
   - помните, что ключевое слово `this` указывает на `GameObject`, к которому прикреплен сценарий. В данном случае это капсула игрока;
   - атрибут `Vector3.forward` умножается на коэффициент `vInput`, а из свойства `Time.deltaTime` берется направление и скорость, с которой капсула движется вперед или назад по оси *z* с рассчитанной нами скоростью;
   - свойство `Time.deltaTime` всегда будет возвращать значение времени в секундах, прошедшее с момента выполнения последнего кадра игры. Обычно это время используется для сглаживания значений, которые захватываются или запускаются в методе `Update`, а не берутся из частоты кадров устройства.
6. Используем метод `Rotate` для поворота компонента `Transform` капсулы относительно вектора, который передаем в качестве параметра:
   - `Vector3.up` умножается на `hInput`, а из свойства `Time.deltaTime` мы берем желаемую ось вращения влево/вправо;
   - все по той же причине мы используем ключевое слово `this` и `Time.deltaTime`.

Как мы обсуждали ранее, использование векторов направления в методах Translate и Rotate — лишь один из способов реализовать движение. Мы могли бы создавать объекты типа Vector3 из входных осей и точно так же применять их в качестве параметров.

Нажав кнопку Play, вы сможете перемещать капсулу вперед и назад с помощью клавиш со стрелками вверх/вниз и клавиш W/S, а также вращать или поворачивать ее с помощью клавиш со стрелками влево/вправо и клавиш A/D. Всего лишь пара строк кода позволила нам создать два отдельных элемента управления, не зависящих от частоты кадров и легко поддающихся настройке. Но камера все еще не следует за капсулой при ее перемещении, поэтому исправим данную оплошность в следующем разделе.

Следование камеры за игроком

Самый простой способ заставить один GameObject следовать за другим — сделать один из них дочерним для другого. Но в этом случае любое движение или вращение, происходящее с капсулой игрока, будет влиять и на камеру, а мы не всегда хотим, чтобы так и было. К счастью, можно без труда задавать положение и поворот камеры относительно капсулы с помощью методов из класса Transform. Сейчас ваша задача — написать логику движения камеры.

Время действовать. Программируем поведение камеры

Поскольку мы хотим, чтобы поведение камеры было полностью отделено от того, как движется игрок, мы будем контролировать положение камеры относительно цели, которую мы можем установить на панели Inspector.

1. Создайте в папке Scripts новый сценарий C#, назовите его CameraBehavior и перетащите его на капсулу Main Camera.
2. Добавьте в сценарий следующий код и сохраните файл:

```
public class CameraBehavior : MonoBehaviour
{
    // 1
    public Vector3 camOffset = new Vector3(0f, 1.2f, -2.6f);

    // 2
    private Transform target;

    void Start()
    {
        // 3
        target = GameObject.Find("Player").transform;
    }

    // 4
    void LateUpdate()
    {
        // 5
        this.transform.position = target.TransformPoint(camOffset);

        // 6
        this.transform.LookAt(target);
    }
}
```

Разберем этот код.

1. Объявляем переменную типа `Vector3`, в которой будем хранить желаемое расстояние между камерой и капсулой игрока:
   - мы сможем вручную установить положения *x*, *y* и *z* смещения камеры на панели Inspector, так как их значения публичные;
   - мне больше всего нравятся именно такие значения по умолчанию, но не стесняйтесь экспериментировать.
2. Создаем переменную для хранения информации о компоненте `Transform` капсулы игрока:
   - это даст нам доступ к атрибутам `posititon`, `rotation` и `scale`;
   - мы не хотим, чтобы эти данные были доступны за пределами сценария `CameraBehavior`, и потому они приватные.
3. Вызываем метод `GameObject.Find`, чтобы найти капсулу по имени и получить ее компонент `Transform` с текущим положением на сцене:
   - это значит, что позиции капсулы по осям *x*, *y* и *z* обновляются и сохраняются в переменной `target` каждый кадр.
4. Элемент `LateUpdate` — это метод `MonoBehavior`, как и `Start` и `Update`, который выполняется после `Update`:
   - поскольку наш сценарий `PlayerBehavior` перемещает капсулу в методе `Update`, код `CameraBehavior` должен выполняться после того, как произошло движение, чтобы в переменной `target` всегда находилось самое актуальное значение.
5. Устанавливаем положение камеры на `target.TransformPoint(camOffset)` в каждом кадре. В результате получается следующее:
   - метод `TransformPoint` вычисляет и возвращает относительное положение в глобальном пространстве;
   - в данном случае он возвращает положение `target` (нашей капсулы), смещенное на `0` по оси *x*, `1.2` по оси *y* (то есть камера над капсулой) и `-2.6` по оси *z* (камера немного позади капсулы).
6. Метод `LookAt` обновляет вращение капсулы каждый кадр, уделяя особое внимание параметру `Transform`, который мы передаем, и в данном случае это `target` (рис. 7.4).

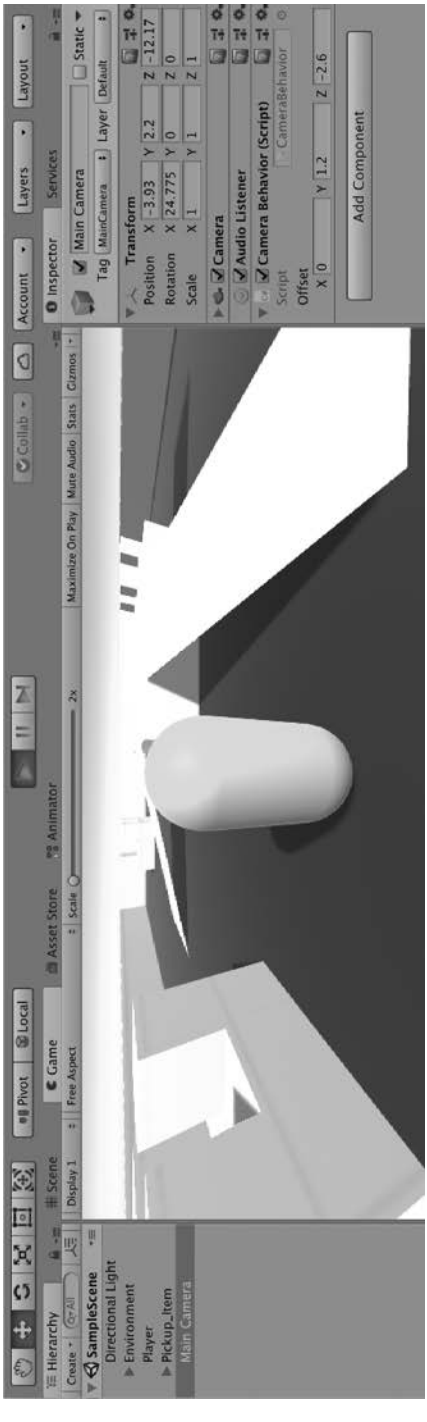

Hierarchy
Create
SampleScene
Directional Light
Environment
Player
Pickup_Item
Main Camera
Scene
Game
Asset Store
Animator
Pivot
Local
Collab
Account
Layers
Layout
Inspector
Services
Main Camera
Static
Tag
MainCamera
Layer
Default
Transform
Position
Rotation
Scale
Camera
Audio Listener
Camera Behavior (Script)
Script
CameraBehavior
Offset
Add Component

Рис. 7.4

Мы рассмотрели множество сложных вещей, но понять происходящее будет легче, если разбить его на хронологические этапы.

1. Мы задали положение смещения для камеры.
2. Мы определяем и сохраняем положение капсулы игрока.
3. Мы вручную обновляем положение и поворот в каждом кадре, чтобы камера всегда следовала на заданном расстоянии и смотрела на игрока.

При использовании методов класса, функциональность которых зависит от платформы, всегда стоит разбивать работу на самые простые этапы. Благодаря этому вы сможете оставаться на плаву, работая в новых средах программирования.

Написанный нами код для управления движением игрока отлично работает, но вы могли заметить, что при движении он иногда слегка дергается. Чтобы создать более плавный и реалистичный эффект движения, необходимо понимать основы системы физики Unity, с которыми вы познакомитесь дальше.

Работа с физикой Unity

До сего момента мы не обсуждали, как работает движок Unity и как он создает реалистичные взаимодействия и движения в виртуальном пространстве. Оставшуюся часть этой главы мы посвятим основам физической системы Unity (рис. 7.5).

Два основных компонента движка NVIDIA PhysX в Unity представлены ниже.

- Компоненты `Rigidbody` позволяют объектам `GameObject` подвергаться воздействию силы тяжести и добавляют объектам такие свойства, как `Mass` и `Drag`. На компоненты `Rigidbody` может воздействовать и приложенная сила, что позволяет реализовать реалистичное движение (рис. 7.6).

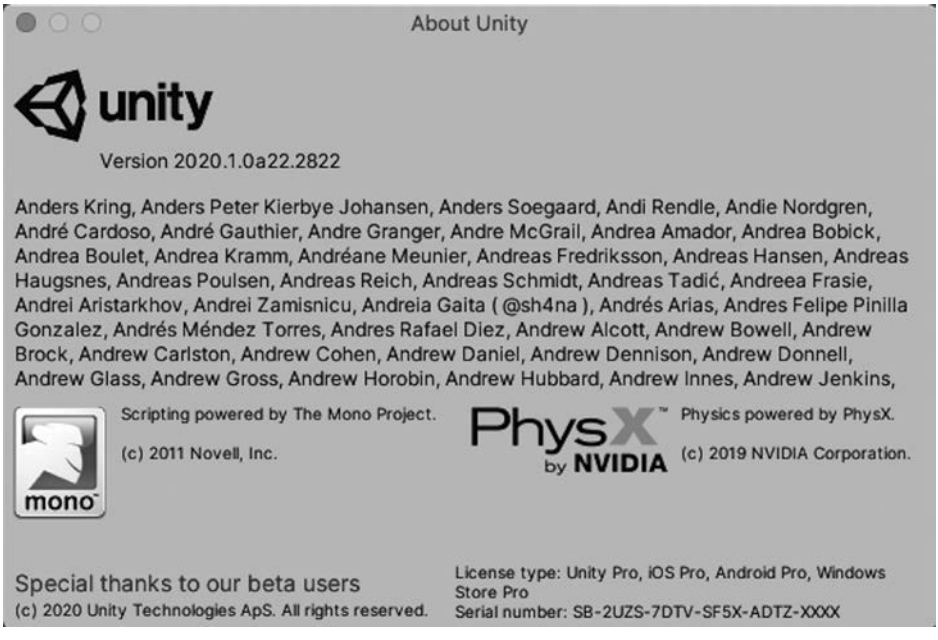

About Unity
unity
Version 2020.1.0a22.2822
Anders Kring, Anders Peter Kierbye Johansen, Anders Soegaard, Andi Rendle, Andie Nordgren, André Cardoso, André Gauthier, Andre Granger, Andre McGrail, Andrea Amador, Andrea Bobick, Andrea Boulet, Andrea Kramm, Andréane Meunier, Andreas Fredriksson, Andreas Hansen, Andreas Haugsnes, Andreas Poulsen, Andreas Reich, Andreas Schmidt, Andreas Tadić, Andreea Frasie, Andrei Aristarkhov, Andrei Zamisnicu, Andreia Gaita (@sh4na), Andrés Arias, Andres Felipe Pinilla Gonzalez, Andrés Méndez Torres, Andres Rafael Diez, Andrew Alcott, Andrew Bowell, Andrew Brock, Andrew Carlston, Andrew Cohen, Andrew Daniel, Andrew Dennison, Andrew Donnell, Andrew Glass, Andrew Gross, Andrew Horobin, Andrew Hubbard, Andrew Innes, Andrew Jenkins,
Scripting powered by The Mono Project.
(c) 2011 Novell, Inc.
mono
PhysX by NVIDIA
Physics powered by PhysX.
(c) 2019 NVIDIA Corporation.
Special thanks to our beta users
(c) 2020 Unity Technologies ApS. All rights reserved.
License type: Unity Pro, iOS Pro, Android Pro, Windows Store Pro
Serial number: SB-2UZS-7DTV-SF5X-ADTZ-XXXX

Рис. 7.5

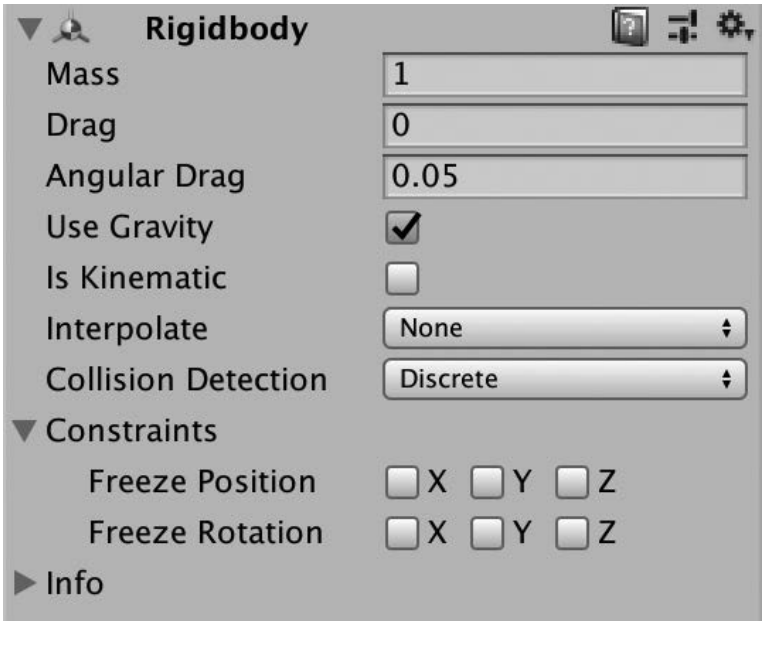

Rigidbody
Mass
1
Drag
0
Angular Drag
0.05
Use Gravity
Is Kinematic
Interpolate
None
Collision Detection
Discrete
Constraints
Freeze Position
X Y Z
Freeze Rotation
X Y Z
Info

Рис. 7.6

- Компоненты `Collider` определяют, как и когда объекты `GameObject` входят и выходят из физического пространства друг друга, сталкиваются или отскакивают. К одному `GameObject` должен быть прикреплен только один компонент `Rigidbody`, а вот компонентов `Collider` может быть несколько. Такой вариант называется составным коллайдером (рис. 7.7).

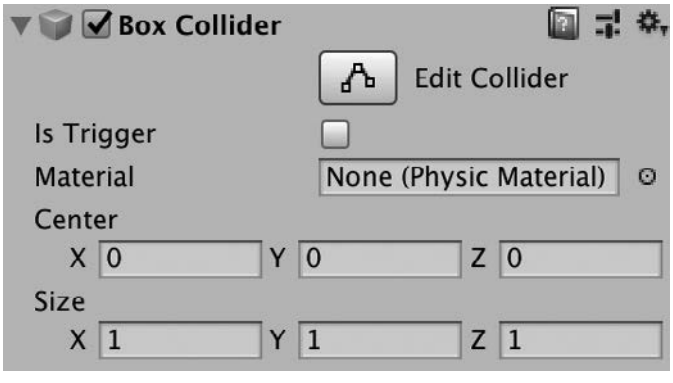


Рис. 7.7

Когда два `GameObject` сталкиваются друг с другом, свойства `Rigidbody` определяют получившееся взаимодействие. Например, если масса одного `GameObject` больше, чем другого, то более легкий `GameObject` будет отталкиваться от крупного с большей силой, как и в реальной жизни. Эти два компонента отвечают за все физические взаимодействия и моделирование движения в Unity.

При использовании этих компонентов нужно проявлять осторожность. Лучше всего рассмотреть этот момент с точки зрения типов движений, которые Unity позволяет задавать.

- *Кинематическое* движение происходит, когда компонент `Rigidbody` прикреплен к `GameObject`, но не регистрируется в физической системе сцены:
  - такое движение используется только в определенных случаях и включается с помощью флажка свойства `Is Kinematic` компонента `Rigidbody`. Мы хотим, чтобы наша капсула взаимодействовала с физической системой, поэтому не будем прибегать к такому движению.

- *Некинематическое* движение — это когда компонент `Rigidbody` перемещается или вращается под действием силы, а не ручного изменения свойств компонента `Transform`. Наша цель в этом разделе — переделать скрипт `PlayerBehavior` и реализовать данный тип движения.

То, что мы реализовали сейчас, а именно манипулирование компонентом Transform капсулы при использовании компонента Rigidbody для взаимодействия с физической системой, было сделано для того, чтобы вы поняли математическую суть движения и вращения в трехмерном пространстве. Однако этот метод не годится для реальной работы, и Unity советует избегать сочетания кинематических и некинематических движений в коде.

Ваша следующая задача — применить силу, чтобы преобразовать текущую систему движения в более реалистичную.

Rigidbody в движении

Поскольку к нашему игроку прикреплен компонент `Rigidbody`, мы можем позволить физическому движку управлять движением, а не крутить параметры компонента `Transform` вручную.

Когда речь идет о применении силы, есть два варианта:

- вы можете сделать это напрямую, используя методы класса `Rigidbody`, такие как `AddForce` и `AddTorque`, которые выполняют перемещение и поворот объекта соответственно. У этого подхода есть свои недостатки, и его применение часто требует дополнительного кода, который компенсирует неожиданное поведение объектов с точки зрения физики;
- в качестве альтернативы вы можете использовать другие методы класса `Rigidbody`, такие как `MovePosition` и `MoveRotation`, которые тоже работают за счет приложенной силы, но решают пограничные случаи без нашего участия.

Ниже мы пойдем по второму пути, но если вам интересно вручную применить силу и крутящий момент к своим объектам GameObject, то вам сюда: docs.unity3d.com/ScriptReference/Rigidbody.AddForce.html.

Оба метода дадут игроку более реалистичное ощущение и позволят нам в главе 8 добавить в игру механику прыжков и рывков.

Если вам интересно, что происходит, когда движущийся объект без компонента Rigidbody взаимодействует с элементами среды, у которых этот компонент есть, то удалите компонент Rigidbody из объекта Player и пробегитесь по арене. Поздравляю — вы привидение и теперь можете проходить сквозь стены! Поэкспериментировав, не забудьте снова добавить компонент Rigidbody!

К капсуле игрока уже прикреплен компонент `Rigidbody`; это значит, что вы можете получить доступ к его свойствам и изменять их. Но для этого нужно сначала найти и сохранить компонент, что и будет вашей следующей задачей.

Время действовать. Получение доступа к компоненту Rigidbody

Вам нужно будет получить и сохранить компонент `Rigidbody` в капсуле игрока, чтобы можно было его изменять.

Измените код сценария `PlayerBehavior` следующим образом:

```
public class PlayerBehavior : MonoBehaviour
{
    public float moveSpeed = 10f;
    public float rotateSpeed = 75f;

    private float vInput;
    private float hInput;

    // 1
    private Rigidbody _rb;
```

```
    // 2
    void Start()
    {
        // 3
        _rb = GetComponent<Rigidbody>();
    }

    void Update()
    {
        vInput = Input.GetAxis("Vertical") * moveSpeed;
        hInput = Input.GetAxis("Horizontal") * rotateSpeed;

        /* 4
        this.transform.Translate(Vector3.forward * vInput *
        Time.deltaTime);
        this.transform.Rotate(Vector3.up * hInput *
            Time.deltaTime);
        */
    }
}
```

Разберем этот код.

1. Добавляем приватную переменную типа `Rigidbody`, которая будет содержать информацию о компоненте `Rigidbody` капсулы.
2. Метод `Start` срабатывает, когда сценарий инициализируется в сцене, то есть в момент нажатия кнопки Play, и должен использоваться каждый раз, когда необходимо задать значения переменных в начале класса.
3. Метод `GetComponent` проверяет, существует ли тип компонента, который мы ищем, в данном случае `Rigidbody`, в `GameObject`, к которому прикреплен сценарий, и возвращает его:
   - если компонент не прикреплен к `GameObject`, то метод вернет `null`, но мы знаем, что он есть, поэтому не будем в данный момент думать о проверке ошибок.
4. Закомментированные вызовы методов `Transform` и `Rotate` в функции `Update`, чтобы мы не запускали два разных типа элементов управления игроком:
   - мы хотим сохранить наш код, который фиксирует ввод со стороны игрока, чтобы использовать его позже.

Вы инициализировали и сохранили компонент `Rigidbody` на капсуле игрока и закомментировали устаревший код `Transform`, чтобы подготовить почву для описания движения на основе физики. Теперь персонаж готов к следующему шагу — добавлению сил.

Время действовать. Перемещаем компонент Rigidbody

Чтобы переместить и повернуть компонент `Rigidbody`, выполните следующие действия.

Добавьте следующий код в сценарий `PlayerBehavior` под методом `Update`, а затем сохраните файл:

```
// 1
void FixedUpdate()
{
    // 2
    Vector3 rotation = Vector3.up * hInput;

    // 3
    Quaternion angleRot = Quaternion.Euler(rotation *
        Time.fixedDeltaTime);

    // 4
    _rb.MovePosition(this.transform.position +
        this.transform.forward * vInput * Time.fixedDeltaTime);

    // 5
    _rb.MoveRotation(_rb.rotation * angleRot);
}
```

Разберем этот код.

1. Любой код, связанный с физикой или `Rigidbody`, всегда находится внутри метода `FixedUpdate`, а не в самом `Update` или в других методах `MonoBehavior`:
   - метод `FixedUpdate` не зависит от частоты кадров и используется для описания всей физики.
2. Создаем новую переменную `Vector3` для хранения вращения влево и вправо:

- результат произведения `Vector3.up * hInput` — тот же вектор вращения, который мы использовали в методе `Rotate` в предыдущем примере.

3. Метод `Quaternion.Euler` принимает на вход `Vector3` и возвращает значение поворота в углах Эйлера:
   - для метода `MoveRotation` нам нужно значение `Quaternion` вместо `Vector3`. Это преобразование в тип вращения, который для Unity предпочтителен;
   - умножение на `Time.fixedDeltaTime` выполняется по той же причине, по которой мы использовали `Time.deltaTime` в методе `Update`.
4. Вызов метода `MovePosition` на нашем компоненте `_rb`, который принимает параметр `Vector3` и соответствующим образом прикладывает силу:
   - используемый вектор можно разбить так: положение `Transform` капсулы в прямом направлении, умноженное на вход вертикальной оси и `Time.fixedDeltaTime`;
   - компонент `Rigidbody` реализует приложение движущей силы, чтобы удовлетворить нашему векторному параметру.
5. Вызов метода `MoveRotation` на нашем компоненте `_rb`, который тоже принимает параметр `Vector3` и соответствующим образом прикладывает силу:
   - в переменной `angleRot` уже содержится ввод горизонтальной оси с клавиатуры, поэтому нам достаточно лишь умножить текущее вращение `Rigidbody` на `angleRot`, чтобы получить одинаковое левое и правое вращение.

Помните, что для некинематических игровых объектов методы MovePosition и MoveRotation работают по-другому. Дополнительную информацию можно найти в Rigidbody Scripting Reference, пройдя по ссылке docs.unity3d.com/ScriptReference/Rigidbody.html.

Если вы нажмете кнопку Play, то сможете двигаться вперед и назад в направлении вашего взгляда, а также поворачивать по оси `y`. Приложенная сила влияет на движение активнее, чем перемещение и вращение ком-

понента Transform, поэтому вам может потребоваться точная настройка переменных moveSpeed и rotateSpeed на панели Inspector. Мы реализовали тот же тип схемы движения, что и раньше, разница лишь в более реалистичной физике.

Если вы забежите по пандусу или спрыгнете с центральной платформы, то увидите, как игрок взлетает в воздух или медленно падает на землю. Несмотря на то что компонент Rigidbody настроен на использование силы тяжести, влияние ее слабо. Мы рассмотрим применение силы тяжести к игроку в следующей главе, когда реализуем механику прыжка. Сейчас у нас другая задача — разобраться, как компоненты Collider обрабатывают столкновения в Unity.

Коллайдеры и столкновения

Компоненты Collider не только позволяют физической системе Unity распознавать объекты, но и дают возможность реализовать взаимодействия и столкновения. Коллайдеры — своего рода невидимые силовые поля, окружающие объекты GameObject. Они могут позволить проходить сквозь них, а могут быть твердыми, в зависимости от их настроек. У коллайдеров есть множество методов, которые срабатывают во время различных взаимодействий.

Физическая система Unity в 2D- и 3D-играх работает по-разному, но в этой книге мы будем рассматривать только 3D. Если вас интересует создание 2D-игр, то прочитайте о компоненте Rigidbody2D и ознакомьтесь с доступными 2D-коллайдерами.

Взгляните на приведенный ниже скриншот объекта Capsule в иерархии объектов Pickup_Prefab (рис. 7.8).

Зеленая оболочка вокруг объекта — это компонент Capsule Collider. Его можно перемещать и масштабировать с помощью свойств Center, Radius и Height. Когда мы работаем с примитивом, его Collider по умолчанию соответствует форме примитива. Поскольку мы создали примитив Capsule, у него в комплекте сразу есть Capsule Collider.

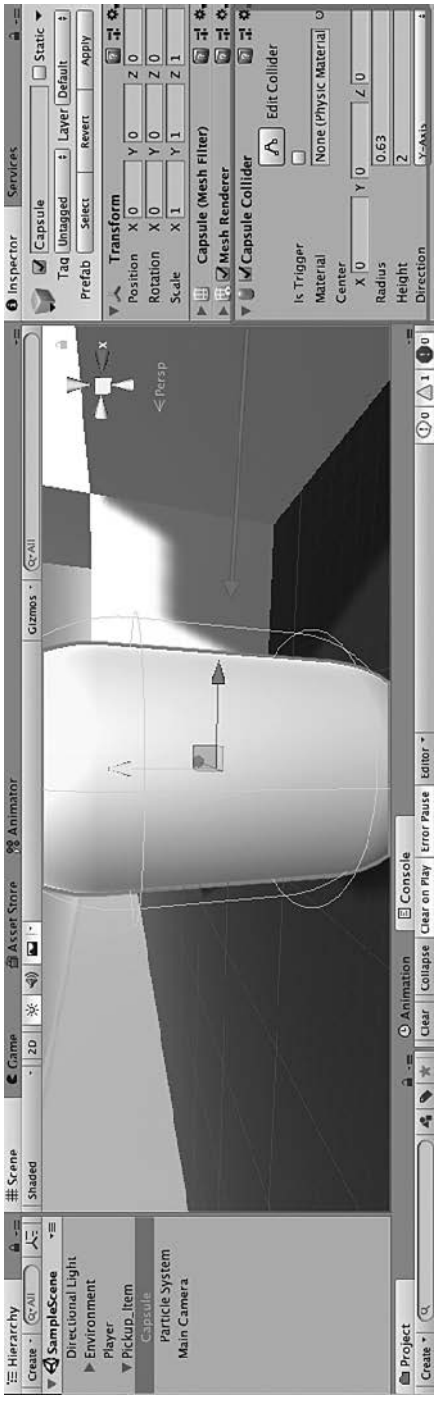

Hierarchy
Create
SampleScene
Directional Light
Environment
Player
Pickup_Item
Capsule
Particle System
Main Camera
Scene
Shaded
2D
Persp
Game
Asset Store
Animator
Gizmos
Project
Create
Animation
Console
Clear
Collapse
Clear on Play
Error Pause
Editor
Inspector
Services
Capsule
Static
Tag
Untagged
Layer
Default
Prefab
Select
Revert
Apply
Transform
Position
Rotation
Scale
Capsule (Mesh Filter)
Mesh Renderer
Capsule Collider
Edit Collider
Is Trigger
Material
None (Physic Material)
Center
Radius
0.63
Height
2
Direction
Y-Axis

Рис. 7.8

У коллайдеров есть различные формы: Box, Sphere и Mesh. Их можно вручную добавлять в меню Component ▸ Physics или с помощью кнопки Add Component на панели Inspector.

Когда компонент `Collider` входит в контакт с другими компонентами, он отправляет так называемое сообщение или рассылку. Любой сценарий, в котором есть один или несколько соответствующих методов, получит уведомление, когда `Collider` отправит сообщение. Это называется *событием*, и о событиях мы поговорим в главе 12.

Например, когда два `GameObject` с коллайдерами контактируют друг с другом, они оба отправляют сообщение `OnCollisionEnter` со ссылкой на объект, с которым произошло столкновение. Данную информацию можно использовать для программирования реакции на столкновение. Самое простое — организовать подбор предмета. Этим и займемся.

Полный список уведомлений, которые выдают компоненты Collider, можно найти в разделе Messages на странице docs.unity3d.com/ScriptReference/Collider.html.

События столкновения и триггера отправляются только в случае, когда сталкивающиеся объекты относятся к определенному набору компонентов Collider, Trigger и RigidBody, а также в случае кинематического или некинематического движения. Подробнее можно почитать в разделе Collision action matrix по адресу docs.unity3d.com/Manual/CollidersOverview.html.

Созданный нами ранее бонус здоровья — идеальный кандидат, на котором мы проверим, как работают столкновения. Приступим.

Время действовать. Подбираем предмет

Чтобы добавить в объект `Pickup_Item` обработку столкновений, необходимо выполнить следующие действия.

1. Создайте новый сценарий C# в папке Scripts, назовите его `ItemBehavior`, а затем перетащите его на объект `Health_Pickup`:
   - любой сценарий, в котором реализовано обнаружение столкновений, *должен* быть прикреплен к `GameObject` с компонентом `Collider`, даже если он является дочерним от префаба.

2. Создайте пустой GameObject с именем Item:
   - сделайте объект Health_Pickup и его Particle System дочерними для этого объекта;
   - перетащите объект Item в папку Prefabs (рис. 7.9).

Рис. 7.9

3. Замените код по умолчанию в сценарии ItemBehavior тем, который представлен ниже, и сохраните его:

```
public class ItemBehavior : MonoBehaviour
{
    // 1
    void OnCollisionEnter(Collision collision)
    {
        // 2
        if(collision.gameObject.name == "Player")
        {
            // 3
            Destroy(this.transform.parent.gameObject);

            // 4
            Debug.Log("Item collected!");
        }
    }
}
```

4. Нажмите кнопку Play и переместите игрока к бонусу, чтобы поднять его!

Разберем этот код.

1. Когда другой объект сталкивается с префабом Item, свойство isTrigger которого включено, Unity автоматически вызывает метод OnCollisionEnter:

- метод `OnCollisionEnter` запускается, принимая в качестве аргумента ссылку на коллайдер, с которым столкнулся объект;
- обратите внимание, что мы задали для столкновения тип `Collision`, а не `Collider`.

2. У класса `Collision` есть свойство `gameObject`, содержащее ссылку на объект, с которым произошло столкновение:
   - через свойство мы можем получить имя объекта и затем использовать оператор `if`, чтобы проверить, является ли этот объект игроком.
3. Если сталкивающимся объектом оказывается игрок, то мы вызываем метод `Destroy()`, который принимает параметр `GameObject`:
   - мы хотим, чтобы весь префаб `Item` был уничтожен;
   - поскольку сценарий `ItemBehavior` прикреплен к объекту `Health_Item`, который является дочерним объектом `Pickup_Item`, мы обращаемся к свойству `this.transform.parent.gameObject` и разрушаем префаб `Item`.
4. Затем мы выводим в консоль простое сообщение о том, что элемент подобран (рис. 7.10).

Мы настроили сценарий `ItemBehavior` внутри префаба `Item` на перехват любых столкновений с объектом `Health_Pickup`. Каждый раз, когда происходит столкновение, `ItemBehavior` вызывает метод `OnCollisionEnter()` и проверяет, является ли второй объект игроком, и если да, то уничтожает (то есть подбирает) элемент. Если вам сложно понять это, то вспомните код столкновения, который мы написали в качестве приемника уведомлений от капсулы `Item`. Каждый раз, когда происходит столкновение, срабатывает код.

Кроме того, важно понимать, что мы могли бы создать аналогичный сценарий с методом `OnCollisionEnter()`, прикрепить его к игроку, а затем проверить, является ли сталкивающийся объект префабом `Item`. Логика столкновения зависит от настроек второго объекта.

Теперь возникает второй вопрос: а как настроить столкновение, не мешая при этом сталкивающимся объектам двигаться сквозь друг друга? Этим вопросом займемся далее.

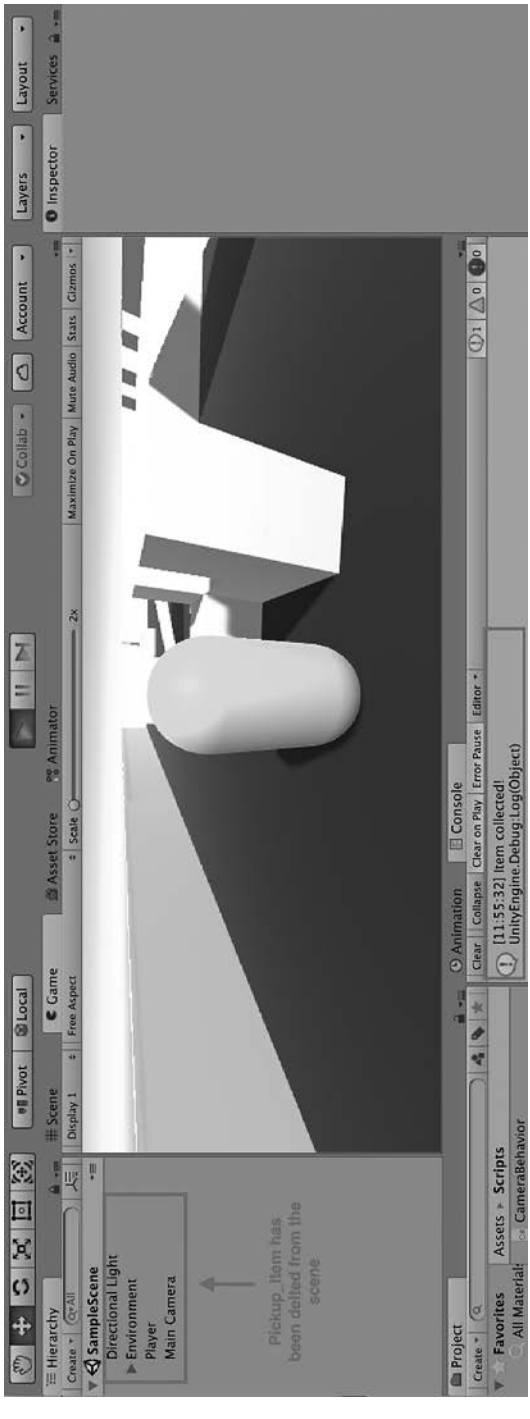

Pivot
Local
Scene
Game
Asset Store
Animator
Collab
Account
Layers
Layout
Hierarchy
Create
Display 1
Free Aspect
Scale
2x
Maximize On Play
Mute Audio
Stats
Gizmos
Inspector
Services
SampleScene
Directional Light
Environment
Player
Main Camera
Pickup_Item has been delted from the scene
Project
Animation
Console
Clear
Collapse
Clear on Play
Error Pause
Editor
Favorites
Assets
Scripts
All Materials
CameraBehavior
[11:55:32] Item collected!
UnityEngine.Debug:Log(Object)

Рис. 7.10

Использование триггеров коллайдера

По умолчанию свойство `isTrigger` у коллайдеров отключено, то есть физическая система рассматривает их как твердые объекты. Однако есть ситуации, когда игроку нужно дать возможность беспрепятственно проходить сквозь компонент `Collider`. Вот здесь и нужны триггеры. Когда свойство `isTrigger` включено, `GameObject` может проходить через коллайдер, но будут вызываться уже другие методы: `OnTriggerEnter`, `OnTriggerExit` и `OnTriggerStay`.

Триггеры наиболее полезны, когда требуется установить, вошел ли `GameObject` в определенную область или проходит через определенную точку. С помощью триггеров мы настроим области вокруг наших врагов. Если игрок войдет в зону срабатывания триггера, то враги узнают об этом и будут нападать на игрока. Но пока займемся логикой поведения врага.

Время действовать. Создаем врага

Чтобы создать врагов на карте, выполните следующие действия.

1. Создайте новый примитив с помощью меню Create ▸ 3D-Object ▸ Capsule на панели Hierarchy и назовите его `Enemy`.
2. В папке Materials используйте команду Create ▸ Material, назовите новый материал `Enemy_Mat` и установите для свойства `Albedo` ярко-красный цвет:
   - перетащите материал `Enemy_Mat` на объект `Enemy`.
3. Выбрав объект `Enemy`, нажмите кнопку Add Component и найдите компонент `Sphere Collider`. Затем нажмите клавишу Enter, чтобы добавить его:
   - установите флажок свойства `isTrigger` и измените свойство `Radius` на `8` (рис. 7.11).

Теперь вокруг нашего врага есть триггер в форме сферы размером в восемь единиц. Каждый раз, когда другой объект будет входить, находиться внутри или выходить из этой области, Unity станет отправлять уведомления о данном событии, которые можно будет перехватить, точно так же, как мы это делали со столкновениями. Следующей вашей задачей будет перехватить уведомление и обработать его в коде.

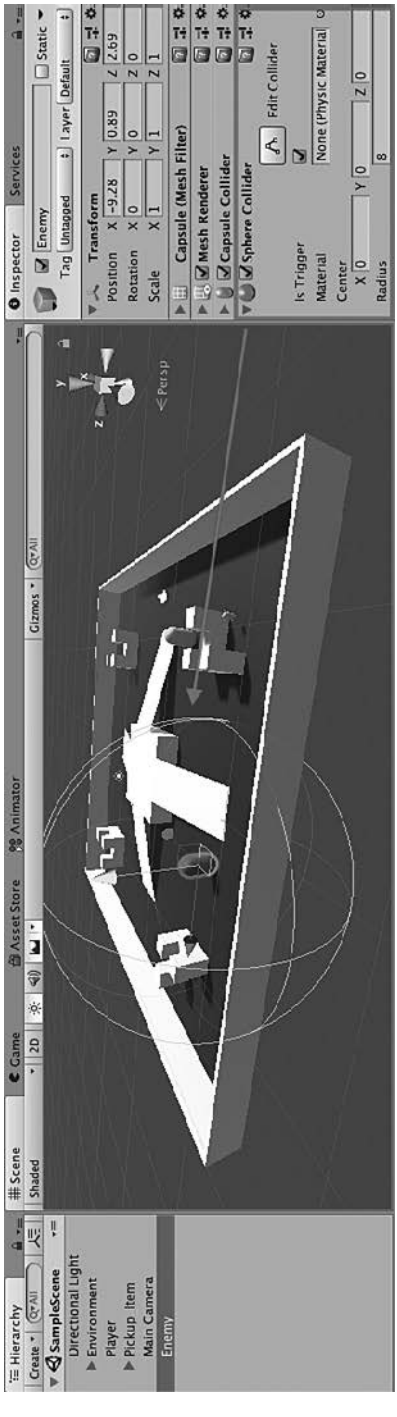

Hierarchy
Create
SampleScene
Directional Light
Environment
Player
Pickup_Item
Main Camera
Enemy
Scene
Shaded
2D
Game
Asset Store
Animator
Gizmos
Persp
Inspector
Services
Enemy
Static
Tag Untagged
Layer Default
Transform
Position X -9.28 Y 0.89 Z 2.69
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Capsule (Mesh Filter)
Mesh Renderer
Capsule Collider
Sphere Collider
Edit Collider
Is Trigger
Material None (Physic Material)
Center X 0 Y 0 Z 0
Radius 8

Рис. 7.11

Время действовать. Перехватываем события триггера

Чтобы перехватить событие триггера, вам необходимо создать новый сценарий, выполнив следующие действия.

1. Создайте новый сценарий C# в папке Scripts, назовите его `EnemyBehaviour`, а затем перетащите его на `Enemy`.
2. Добавьте в сценарий следующий код и сохраните файл:

```
public class EnemyBehavior : MonoBehaviour
{
    // 1
    void OnTriggerEnter(Collider other)
    {
        //2
        if(other.name == "Player")
        {
            Debug.Log("Player detected - attack!");
        }
    }

    // 3
    void OnTriggerExit(Collider other)
    {
        // 4
        if(other.name == "Player")
        {
            Debug.Log("Player out of range, resume patrol");
        }
    }
}
```

3. Нажмите кнопку Play и подойдите к врагу. Появится первое уведомление:
   - затем отойдите от врага, чтобы активировать второе уведомление.

Разберем этот код.

1. Метод `OnTriggerEnter()` срабатывает всякий раз, когда объект входит в область `Sphere Collider` объекта `Enemy`:
   - как и метод `OnCollisionEnter()`, `OnTriggerEnter()` хранит ссылку на компонент `Collider` второго объекта;
   - обратите внимание, что атрибут `other` относится к типу `Collider`, а не `Collision`.

2. Мы можем использовать атрибут `other`, чтобы получить имя второго сталкивающегося `GameObject` и проверить, `Player` это или нет:
   - если это он, то в консоли выводится сообщение о том, что игрок попал в опасную зону.
3. Метод `OnTriggerExit()` срабатывает, когда объект покидает коллайдер объекта `Enemy`:
   - этот метод тоже содержит ссылку на `Collider` второго сталкивающегося объекта (рис. 7.12).
4. Проверяем имя объекта, выходящего из `Sphere Collider`, с помощью оператора `other`:
   - если это `Player`, то выводим сообщение о том, что теперь игрок в безопасности (рис. 7.13).

Область `Sphere Collider` на объекте `Enemy` рассылает уведомления, когда в его области что-то происходит, а сценарий `Enemy Behavior` обрабатывает события входа и выхода. Каждый раз, когда игрок входит в радиус коллайдера или выходит из него, в консоли появляется сообщение о том, что код работает. Мы разовьем эту тему в главе 9.

В Unity используется так называемый паттерн проектирования Component. Не вдаваясь в подробности, смысл этого подхода в том, что объекты (и в более широком смысле их классы) сами обрабатывают свое поведение. Вот почему у нас созданы отдельные сценарии столкновения для бонуса и врага, а не реализовано все в одном классе. Подробнее об этом — в главе 12.

Поскольку в книге мы стремимся показать вам как можно больше хороших приемов и навыков программирования, ваша последняя задача в этой главе — преобразовать все основные объекты в префабы.

Испытание героя. Трубим общий сбор!

Чтобы подготовить проект к следующей главе, перетащите объекты `Player` и `Enemy` в папку Prefabs. Помните: с настоящего момента вам всегда нужно будет нажимать кнопку Apply на панели Inspector, чтобы закрепить любые изменения, которые вы будете вносить в эти объекты `GameObject`.

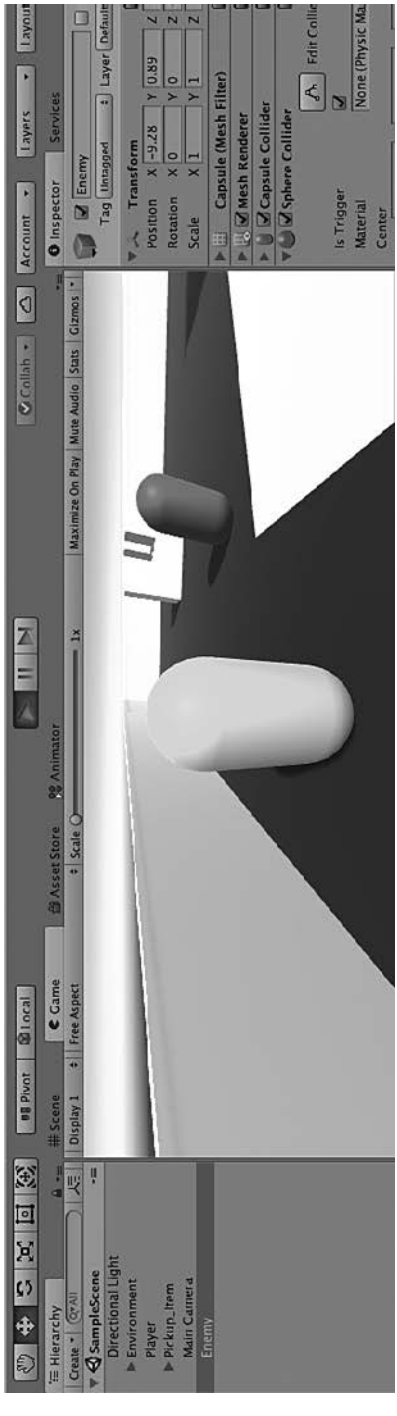

Hierarchy
Create
SampleScene
Directional Light
Environment
Player
Pickup_Item
Main Camera
Enemy
Pivot
Local
Scene
Game
Asset Store
Animator
Display 1
Free Aspect
Scale
1x
Maximize On Play
Mute Audio
Stats
Gizmos
Collab
Account
Layers
Inspector
Services
Enemy
Tag
Untagged
Layer
Transform
Position
Rotation
Scale
Capsule (Mesh Filter)
Mesh Renderer
Capsule Collider
Sphere Collider
Edit Collid
Is Trigger
Material
Center
None (Physic Ma

Рис. 7.12

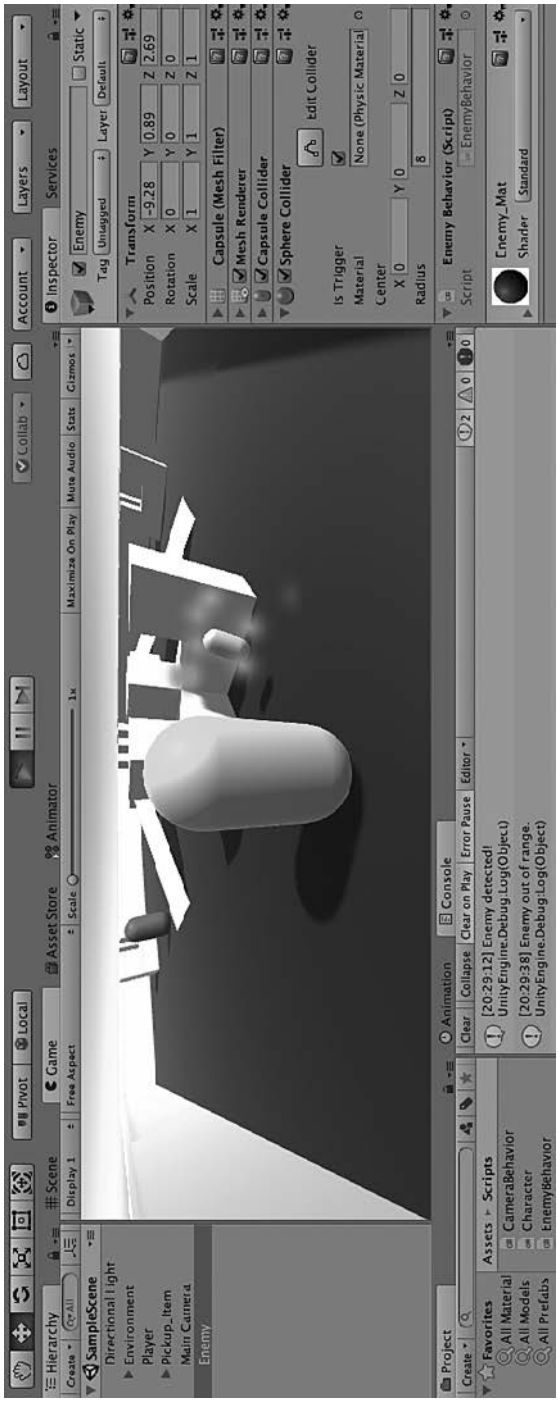

Hierarchy
Create
SampleScene
Directional light
Environment
Player
Pickup_Item
Main Camera
Enemy
Pivot
Local
Scene
Game
Asset Store
Animator
Collab
Account
Layers
Layout
Display 1
Free Aspect
Scale
1x
Maximize On Play
Mute Audio
Stats
Gizmos
Inspector
Services
Enemy
Static
Tag
Untagged
Layer
Default
Transform
Position
Rotation
Scale
Capsule (Mesh Filter)
Mesh Renderer
Capsule Collider
Sphere Collider
Edit Collider
Is Trigger
Material
None (Physic Material)
Center
Radius
Enemy Behavior (Script)
Script
EnemyBehavior
Enemy_Mat
Shader
Standard
Project
Create
Favorites
All Material
All Models
All Prefabs
Assets
Scripts
CameraBehavior
Character
EnemyBehavior
Animation
Console
Clear
Collapse
Clear on Play
Error Pause
Editor
[20:29:12] Enemy detected!
UnityEngine.Debug:Log(Object)
[20:29:38] Enemy out of range.
UnityEngine.Debug:Log(Object)

Рис. 7.13

После этого переходите к следующему подразделу и повторите все основные темы, которые мы рассмотрели, прежде чем двигаться дальше.

Итоги по физике

Прежде чем мы завершим главу, обобщим несколько важных моментов и тем самым закрепим все, что мы узнали к настоящему моменту.

- Компоненты `Rigidbody` добавляют `GameObject`, к которому они прикреплены, правила физики реального мира.
- Компоненты `Collider` взаимодействуют друг с другом, а также с объектами, используя компоненты `Rigidbody`:
  - если компонент `Collider` не является триггером, то работает как твердый объект;
  - если компонент `Collider` является триггером, то сквозь него можно ходить.
- Объект является *кинематическим*, если у него есть компонент `Rigidbody` и включено свойство `Is Kinematic`, которое дает физической системе указание игнорировать его.
- Объект *не является кинематическим*, если у него есть компонент `Rigidbody` и его движение и вращение реализованы через работу с силой и моментом.
- Коллайдеры отправляют уведомления при наступлении некоторых событий:
  - эти уведомления зависят от того, активирован ли компонент `Collider`;
  - уведомления направляются любым из сталкивающихся объектов и содержат ссылку с информацией о другом объекте.

Помните: такая обширная и сложная тема, как система физики Unity, не изучается за день. Полученные в этой главе знания могут помочь перейти к более сложным темам!

Подведем итоги

В этой главе вы приобрели первый опыт создания независимого игрового поведения и объединения разных объектов в одну простую игру. Мы использовали векторы и базовую векторную математику для определения положения и углов в трехмерном пространстве, вы познакомились с методикой ввода данных со стороны игрока и двумя основными методами перемещения и поворота игровых объектов. Мы даже погрузились в тонкости физической системы Unity, поработали с физикой `Rigidbody`, коллайдерами, триггерами и уведомлениями о событиях. В общем, у нашей игры Hero Born отличный старт.

В главе 8 мы начнем заниматься другими игровыми механиками и реализуем прыжки, рывок, стрельбу и взаимодействие с компонентами окружающей среды. Это позволит вам получить больше практического опыта по части применения силы к компонентам `Rigidbody`, сбора данных от игроков и реализации логики нужного нам поведения.

Контрольные вопросы. Управление игроком и физика

1. Какой тип данных используется для хранения информации о трехмерном движении и вращении?
2. Какой встроенный компонент Unity позволяет отслеживать и изменять элементы управления игроком?
3. Какой компонент добавляет к `GameObject` физику реального мира?
4. Какой метод Unity позволяет выполнять с объектами `GameObject` код, связанный с физикой?

8 Программируем механику игры

В предыдущей главе мы в основном занимались кодом, реализующим перемещение игрока и камеры, и немного поговорили о физике Unity. Но для создания интересной игры недостаточно только хождения по уровню. Хотя этот элемент, вероятно, можно считать универсальным почти для всех жанров.

Изюминка игры обычно заключается в ее основной механике и в ощущении своей власти над миром и свободы действия, которую эта механика дает игроку. Без интересных способов воздействия на созданный вами виртуальный мир у вашей игры нет шансов на повторное прохождение игроком, не говоря уже об удовольствии для него. А пока мы будем реализовывать механику игры, параллельно с этим разовьем наши знания о языке C# и его более сложных функциях.

В этой главе мы доделаем прототип игры Hero Born, сосредоточив основное внимание на реализации игровой механики, а также займемся основами системного дизайна и разработки пользовательских интерфейсов (user interfaces, UI). Мы изучим следующие темы:

- добавление прыжков;
- основы работы со слоями-масками;
- создание экземпляров объектов и префабов;
- класс менеджера игры;
- свойства для чтения значения и его установки;
- хранение информации об очках;
- программирование пользовательского интерфейса.

Добавление прыжков

При использовании компонента `Rigidbody` для управления движением игрока мы получаем огромное преимущество: можно легко добавлять различные механики, зависящие от приложенной к объекту силы, например прыжки. В этом разделе мы научим игрока прыгать и напишем нашу первую вспомогательную функцию.

Вспомогательная функция — метод класса, который выполняет рутинную работу, не усложняя при этом код игрового процесса. В такой функции мы можем, например, проверить, касается ли капсула игрока земли (чтобы узнать, можно ли прыгать).

Для начала вам необходимо познакомиться с новым типом данных, называемым перечислениями.

Перечисления

По определению, тип *«перечисление»* — это набор или коллекция именованных констант, объединенный под одним именем переменной. Перечисления полезны, когда вам требуется набор разных значений, но, кроме этого, перечисления относятся к общему родительскому типу, что является дополнительным преимуществом.

Работу с перечислениями проще показать, чем описать на словах, поэтому рассмотрим их синтаксис в следующем фрагменте кода:

```
enum PlayerAction { Attack, Defend, Flee };
```

Разберем этот код.

- Ключевое слово `enum` объявляет тип, а затем указывается имя этого типа.
- Значения, которые входят в перечисление, перечисляются в фигурных скобках через запятую (кроме последнего элемента).
- Оператор `enum` должен заканчиваться точкой с запятой, как и все другие типы данных, с которыми мы работали.

Чтобы объявить переменную перечисления, используется следующий синтаксис:

```
PlayerAction currentAction = PlayerAction.Defend;
```

Опять же поясню, как здесь все работает.

- Тип переменной — `PlayerAction`.
- Имя переменной и ее значение, взятое из `PlayerAction`.
- Доступ к каждому значению из перечисления можно получить через точечную нотацию.

На первый взгляд перечисления кажутся невероятно простыми, но при правильном применении они чрезвычайно эффективны. Одна из наиболее полезных черт перечислений — возможность хранить базовые типы. Об этом и поговорим далее.

Основные типы

У перечислений, кроме прочего, есть базовый тип; это значит, что каждая константа, указанная в фигурных скобках, имеет связанное с ней значение. Базовый тип по умолчанию — `int`, и счет начинается с `0`, как и в массивах, причем каждой следующей константе присваивается следующий по величине номер.

Не все типы созданы равными — базовыми типами для перечислений могут быть byte, sbyte, short, ushort, int, uint, long и ulong. Это целочисленные типы, и от выбора типа зависит величина числовых значений, которые может хранить переменная.

Это уже довольно сложный вопрос для данной книги, но в большинстве случаев в реальной жизни достаточно использовать int.

Более подробную информацию об этих типах можно найти здесь: docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/enum.

Например, в нашем перечислении `PlayerAction` в данный момент содержатся следующие значения (хотя явно мы туда их не вносили):

```
enum PlayerAction { Attack = 0, Defend = 1, Flee = 2 };
```

Нет никакого правила, согласно которому базовые значения должны начинаться с `0`. Фактически мы можем сами указать первое значение, а затем C# будет автоматически увеличивать остальные значения, как показано в следующем фрагменте кода:

```
enumPlayerAction { Attack = 5, Defend, Flee };
```

В предыдущем примере константа `Defend` автоматически приняла значение `6`, а `Flee` — `7`. Однако если мы не хотим, чтобы в перечислении `PlayerAction` были последовательные значения, то могли бы задать их вручную:

```
enum PlayerAction { Attack = 10, Defend = 5, Flee = 0};
```

Мы даже можем заменить базовый тип `PlayerAction` на любой из возможных типов, добавив двоеточие после имени перечисления, как показано ниже:

```
enum PlayerAction : byte { Attack, Defend, Flee };
```

Для получения базового типа перечисления требуется явное преобразование, но мы уже рассмотрели его, поэтому приведенный ниже код уже не должен вызывать удивления:

```
enum PlayerAction { Attack = 10, Defend = 5, Flee = 0};

PlayerAction currentAction = PlayerAction.Attack;
int actionCost = (int)currentAction;
```

Перечисления — это чрезвычайно эффективный инструмент вашего арсенала программирования. Ваша следующая задача — применить вновь обретенные знания о перечислениях, чтобы получить данные, вводимые пользователем с клавиатуры.

Время действовать. Нажимаем пробел, чтобы прыгнуть!

Теперь, понимая основы работы перечислений, мы можем перехватить ввод с клавиатуры с помощью перечисления `KeyCode`.

Добавьте в сценарий `PlayerBehavior` следующий код, сохраните его и нажмите кнопку Play:

```
public class PlayerBehavior : MonoBehaviour
{
    public float moveSpeed = 10f;
    public float rotateSpeed = 75f;

    // 1
    public float jumpVelocity = 5f;

    private float vInput;
    private float hInput;

    private Rigidbody _rb;

    void Start()
    {
        _rb = GetComponentRigidbody>();
    }

    void Update()
    {
        // ... Изменения не требуются ...
    }

    void FixedUpdate()
    {
        // 2
        if(Input.GetKeyDown(KeyCode.Space))
        {
            // 3
            _rb.AddForce(Vector3.up * jumpVelocity, ForceMode.Impulse);
        }

        // ... Другие изменения не требуются ...
    }
}
```

Разберем этот код.

1. Мы создаем новую переменную, в которой будем хранить желаемую величину силы прыжка, которую затем можно будет настроить на панели Inspector.
2. Метод `Input.GetKeyDown()` возвращает значение `bool`, зависящее от того, нажата ли указанная клавиша.
   - Метод принимает ключевой параметр типа `string` или `KeyCode`, который является перечислением. Мы указываем, что хотим проверить, нажата ли клавиша с кодом: `KeyCode.Space`.

- Мы используем оператор `if` для проверки возвращаемого значения метода. Если метод вернул `true`, то выполняем тело оператора.

3. Поскольку у нас уже есть компонент `Rigidbody`, мы можем передать ему параметры `Vector3` и `ForceMode` для метода `RigidBody.AddForce()`, тем самым заставив игрока прыгать.
   - Мы указываем, что вектор (или приложенная сила) должен быть направлен вверх, и его величина умножается на параметр `jumpVelocity`.
   - Параметр `ForceMode` определяет, как применяется сила, и также является типом перечисления. Параметр `Impulse` передает объекту мгновенную силу с учетом его массы, что идеально подходит для реализации механики прыжка.

Другие варианты ForceMode могут быть полезны в различных ситуациях, которые подробно описаны здесь: docs.unity3d.com/ScriptReference/ForceMode.html.

Запустив игру прямо сейчас, вы сможете ходить и прыгать, нажимая пробел. Но в данный момент код позволяет вам прыгать бесконечно, а нам бы этого не хотелось. В следующем разделе мы постараемся ограничить нашу механику прыжков, чтобы они не накладывались друг на друга. Для этого воспользуемся так называемым слоем-маской.

Работа со слоями-масками

Слои-маски — это невидимые группы, к которым может принадлежать `GameObject`, используемые физической системой для определения самых разных взаимодействий, от навигации до пересекающихся компонентов коллайдера. Продвинутая работа со слоями-масками выходит за рамки этой книги, но мы создадим простую маску и применим ее, чтобы проверить, стоит ли игрок на земле.

Время действовать. Настраиваем слои объектов

Чтобы можно было проверить, касается ли капсула игрока земли, нам нужно добавить все объекты на нашем уровне на слой-маску. Это позволит нам реализовать вычисление столкновений с компонентом `Capsule Collider`, который уже прикреплен к игроку.

Выполните следующие действия.

1. Выберите любой `GameObject` на панели Hierarchy и выберите команду Layer ▸ Add Layer, как показано на рис. 8.1.

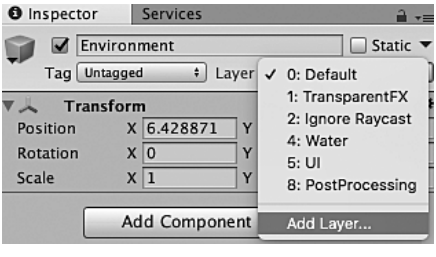


Рис. 8.1

2. Добавьте новый слой под названием `Ground`, введя это имя в первый доступный слот, как показано на рис. 8.2.

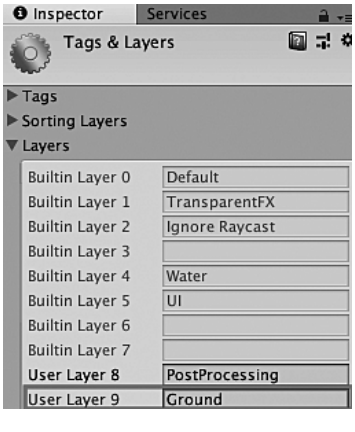


Рис. 8.2

3. Выберите родительский объект `Environment` на панели Hierarchy, щелкните на выпадающем списке Layer и выберите в нем слой Ground. Выбрав слой Ground, показанный на рис. 8.3, нажмите кнопку Yes, когда появится диалоговое окно с вопросом о том, хотите ли вы изменить все дочерние объекты.

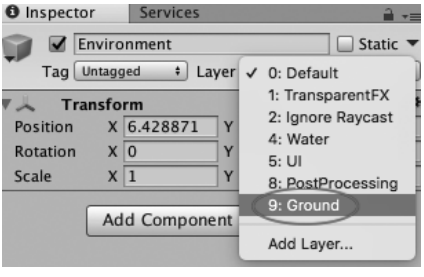


Рис. 8.3

По умолчанию слои 0–7 заняты движком Unity, а нам остается 24 слоя под наши нужды. Мы определили новый слой Ground и назначили ему все объекты группы `Environment`. В дальнейшем все объекты на этом слое можно будет проверить на предмет касания с конкретным объектом. Это мы и сделаем в следующем задании, чтобы убедиться, что игрок может совершить прыжок, только стоя на земле (тем самым устраним проблему бесконечных прыжков).

Время действовать. Прыгаем, но не взлетаем

Поскольку мы не хотим, чтобы код загромождал метод `Update()`, вычисления, связанные с маской, будем выполнять в служебной функции и затем вернем значение `true` или `false` в зависимости от результата. Для этого выполните следующие действия.

1. Добавьте в сценарий `PlayerBehavior` следующий код и снова запустите сцену:

```
public class PlayerBehavior : MonoBehaviour
{
    public float moveSpeed = 10f;
```

```
public float rotateSpeed = 75f;
public float jumpVelocity = 5f;

// 1
public float distanceToGround = 0.1f;

// 2
public LayerMask groundLayer;

private float _vInput;
private float _hInput;
private Rigidbody _rb;

// 3
private CapsuleCollider _col;

void Start()
{
    _rb = GetComponent<Rigidbody>();

    // 4
    _col = GetComponent<CapsuleCollider>();
}

void Update()
{
    // ... Изменения не требуются ...
}

void FixedUpdate()
{
    // 5
    if(IsGrounded() && Input.GetKeyDown(KeyCode.Space))
    {
        _rb.AddForce(Vector3.up * jumpVelocity,
        ForceMode.Impulse);
    }

    // ... Другие изменения не требуются ...
}

// 6
private bool IsGrounded()
{
```

```
        // 7
        Vector3 capsuleBottom = new Vector3(_col.bounds.center.x,
        _col.bounds.min.y, _col.bounds.center.z);

        // 8
        bool grounded = Physics.CheckCapsule(_col.bounds.
        center, capsuleBottom, distanceToGround, groundLayer,
        QueryTriggerInteraction.Ignore);

        // 9
        return grounded;
    }
}
```

2. Установите для параметра `Ground Layer` на панели Inspector значение `Ground` из выпадающего списка, как показано на рис. 8.4.

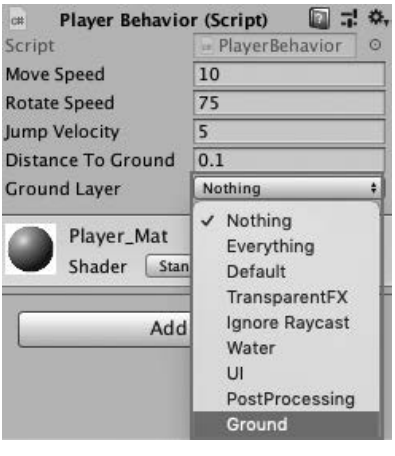


Рис. 8.4

Разберем этот код.

1. Мы создаем новую переменную, которую будем использовать для проверки расстояния между `Capsule Collider` игрока и любым объектом со слоя Ground.
2. Мы создаем переменную `LayerMask`, которую можем настроить на панели Inspector и использовать для обнаружения коллайдера.
3. Мы создаем переменную для хранения `Capsule Collider` игрока.

4. С помощью метода `GetComponent()` найдем и вернем `Capsule Collider`, прикрепленный к игроку.
5. Перепишем оператор `if`, чтобы проверить, возвращает ли атрибут `IsGrounded true` и что перед выполнением кода прыжка нажимается пробел.
6. Объявляем метод `IsGrounded`, который возвращает тип `bool`.
7. Создаем локальную переменную `Vector3`, в которой будем хранить позиции нижней точки коллайдера игрока, и станем использовать ее для проверки на столкновения с любыми объектами на слое Ground.
   - Все компоненты `Collider` имеют свойство `bounds`, которое дает нам доступ к минимальному, максимальному и центральному положению осей *x*, *y* и *z*.
   - Нижняя часть коллайдера — это точки с координатами `center x`, `min y` и `center z`.
8. Создаем локальную переменную типа `bool`, в которой будем хранить результат вызова метода `CheckCapsule()` класса `Physics`. Этот метод принимает следующие пять аргументов:
   - начало капсулы, которое у нас находится в середине `Capsule Collider`, поскольку нас интересует только то, касается ли низ капсулы земли;
   - конец капсулы, который находится в уже рассчитанной нами позиции `capsuleBottom`;
   - радиус капсулы, равный `distanceToGround`, который уже установлен;
   - слой-маска, на котором мы хотим проверять столкновения, — в нашем случае это `GroundLayer` на панели Inspector;
   - описание взаимодействия с триггером запроса, которое определяет, должен ли метод игнорировать коллайдеры, отмеченные как триггеры. Поскольку мы хотим игнорировать все триггеры, мы использовали перечисление `QueryTriggerInteraction.Ignore`.
9. В конце вычисления мы возвращаем значение, хранящееся в переменной `grounded`.

Мы могли бы выполнить расчет столкновений и вручную, но это потребовало бы более сложной трехмерной математики, а у нас нет столько времени, чтобы реализовать все это. Но помните, что всегда рекомендуется использовать встроенные методы, если они есть.

Мы добавили в сценарий `PlayerBehavior` довольно сложный фрагмент кода, но, разобрав его, мы понимаем: единственное, что мы сделали, — это использовали метод из класса `Physics`. Говоря простым языком, предоставили методу `CheckCapsule()` начальную и конечную точки, радиус столкновения и маску. Если конечная точка приближается к объекту на слое-маске на расстояние ближе, чем радиус столкновения, то метод возвращает `true`; это значит, что игрок касается земли. Если игрок находится в процессе прыжка, то метод `CheckCapsule()` возвращает `false`. Поскольку мы проверяем параметр `IsGround` в операторе `if` в каждом кадре метода `Update()`, игрок сможет прыгать, только когда стоит на земле.

На этом мы закончили работу с механикой прыжка, но игроку по-прежнему нужен способ взаимодействия и защиты от полчищ врагов, которые рано или поздно заполонят арену. В следующем разделе мы исправим данную оплошность, реализовав простую механику стрельбы.

Реализация стрельбы

Механика стрельбы настолько распространена, что трудно представить игру от первого лица без какой-либо ее реализации, и Hero Born не исключение. В этом разделе мы поговорим о том, как создавать экземпляры объектов из префабов прямо во время работы игры, и используем полученные навыки, чтобы снаряды двигались вперед под действием физики Unity.

В предыдущих главах вы создавали объекты с помощью конструкторов классов. Создание экземпляров объектов изнутри Unity работает несколько иначе, и об этом поговорим в следующем подразделе.

Создание экземпляров объектов

Концепция создания экземпляра `GameObject` в игре аналогична созданию экземпляра класса. В обоих методах требуется начальное значение, чтобы C# знал, какой тип объекта мы хотим создать и где он нам нужен. Однако, создавая экземпляр `GameObject` на сцене, мы можем оптимизировать процесс, используя метод `Instantiate()` и передав ему префаб, желаемое начальное положение и поворот.

По сути, мы тем самым говорим Unity создать выбранный объект со всеми его компонентами и сценариями в данном конкретном месте и с конкретным направлением поворота, а затем сможем манипулировать им так, как нам захочется, когда он будет существовать в трехмерном пространстве. Прежде чем мы создадим экземпляр объекта, вам необходимо создать сам префаб, чем и займемся.

Время действовать. Создаем префаб снаряда

Прежде чем мы научимся стрелять какими-либо снарядами, нам понадобится ссылка на префаб, поэтому создадим ее прямо сейчас.

1. Выберите команду Create ▸ 3D Object ▸ Sphere на панели Hierarchy и назовите новый объект `Bullet`.
   - Измените масштаб пули в компоненте `Transform` на `0,15` по осям `x`, `y` и `z`.
2. Используйте кнопку Add Component, найдите и добавьте компонент `Rigidbody`, оставив все его свойства по умолчанию.
3. Создайте новый материал в папке Materials с помощью команды Create ▸ Material и назовите его `Orb_Mat`:
   - измените свойство `Albedo` на темно-желтый;
   - перетащите материал на объект `Bullet`.
4. Перетащите объект `Bullet` в папку Prefabs и удалите ее с панели Hierarchy следующим образом (рис. 8.5).

Теперь мы создали и настроили `GameObject` для префаба пули, который можно создавать в игре сколько угодно раз, а также управлять его свойствами желаемым для нас образом. Это значит, что теперь мы готовы перейти к следующей задаче — к стрельбе этими снарядами.

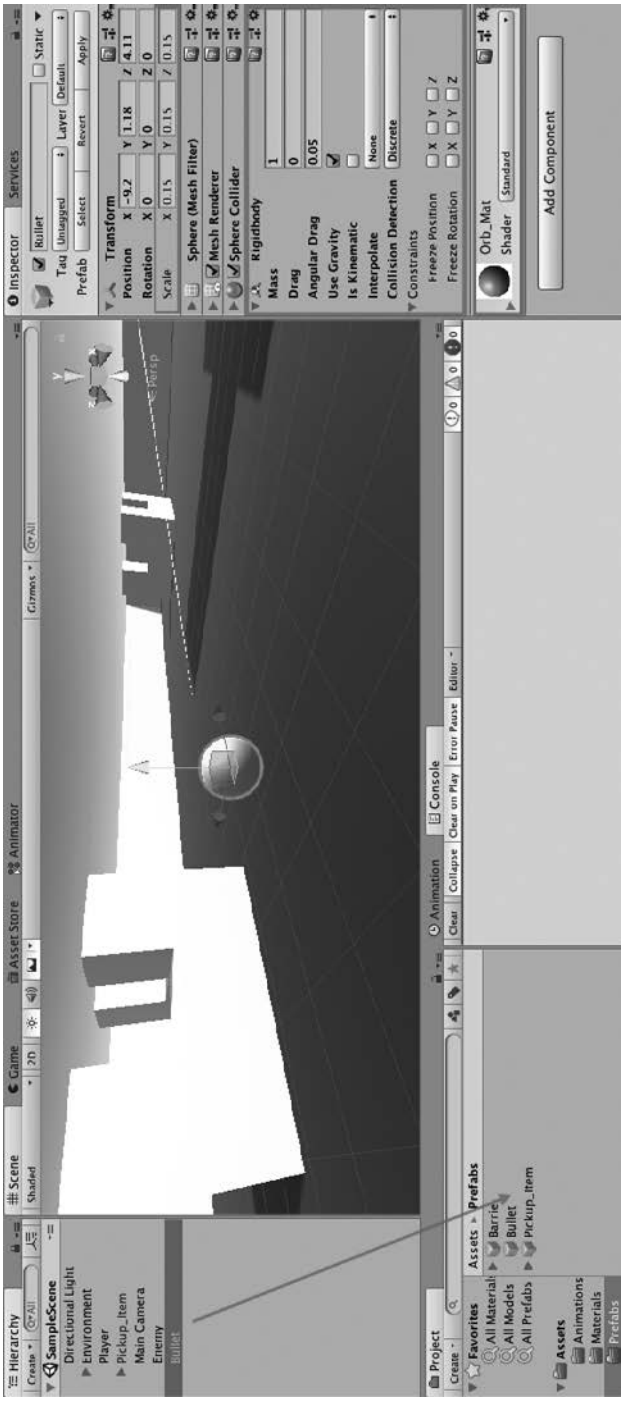

Hierarchy
Scene
Game
Asset Store
Animator
Inspector
Services
SampleScene
Directional Light
Environment
Player
Pickup_Item
Main Camera
Enemy
Bullet
Persp
Project
Animation
Console
Clear
Collapse
Clear on Play
Error Pause
Editor
Favorites
All Materials
All Models
All Prefabs
Assets
Animations
Materials
Prefabs
Barrel
Bullet
Pickup_Item
Transform
Position
Rotation
Scale
Sphere (Mesh Filter)
Mesh Renderer
Sphere Collider
Rigidbody
Mass
Drag
Angular Drag
Use Gravity
Is Kinematic
Interpolate
Collision Detection
Constraints
Freeze Position
Freeze Rotation
Orb_Mat
Shader
Standard
Add Component

Рис. 8.5

Время действовать. Добавляем механику стрельбы

Теперь, когда у нас есть префаб, с которым можно работать, мы можем создавать и перемещать его копии всякий раз, когда игрок нажимает левую кнопку мыши. Это и будет механика стрельбы.

1. Добавьте в сценарий `PlayerBehavior` следующий код:

```
public class PlayerBehavior : MonoBehaviour
{
    public float moveSpeed = 10f;
    public float rotateSpeed = 75f;
    public float jumpVelocity = 5f;
    public float distanceToGround = 0.1f;
    public LayerMask groundLayer;

    // 1
    public GameObject bullet;
    public float bulletSpeed = 100f;

    private float _vInput;
    private float _hInput;
    private Rigidbody _rb;
    private CapsuleCollider _col;

    void Start()
    {
        // ... Изменения не требуются ...
    }

    void Update()
    {
        // ... Изменения не требуются ...
    }

    void FixedUpdate()
    {
        // ... Другие изменения не требуются ...

        // 2
        if (Input.GetMouseButtonDown(0))
        {
            // 3
            GameObject newBullet = Instantiate(bullet,
```

```
                this.transform.position + new Vector3(1, 0, 0),
                this.transform.rotation) as GameObject;

                // 4
                Rigidbody bulletRB = newBullet.GetComponent<Rigidbody>();

                // 5
                bulletRB.velocity = this.transform.forward * bulletSpeed;
            }
        }

        private bool IsGrounded()
        {
            // ... Изменения не требуются ...
        }
    }
```

2. Перетащите префаб `Bullet` в свойство `bullet` объекта `PlayerBehavior` на панели Inspector, как показано на рис. 8.6.
3. Запустите игру и попробуйте понажимать левую кнопку мыши. Капсула начнет стрелять снарядами в направлении, в котором смотрит игрок!

Разберем этот код.

1. Мы создали две переменные: одну для хранения префаба `Bullet`, другую — для хранения скорости пули.
2. Используем оператор `if`, чтобы проверить, возвращает ли метод `Input.GetMouseButtonDown()` значение `true`, как делали ранее с методом `Input.GetKeyDown()`.

   Метод `GetMouseButtonDown()` принимает параметр `int`, определяющий, какую кнопку мыши мы хотим проверить: `0` — левая кнопка, `1` — правая кнопка, а `2` — средняя кнопка или колесо.

Проверка ввода со стороны пользователя в методе FixedUpdate иногда может привести к потере ввода или даже к двойному вводу, поскольку выполняется не точно в каждом кадре. Мы собираемся оставить код как есть, чтобы он остался простым, но другим решением будет проверка входных данных в методе Update, а силу и скорость станем задавать в методе FixedUpdate.

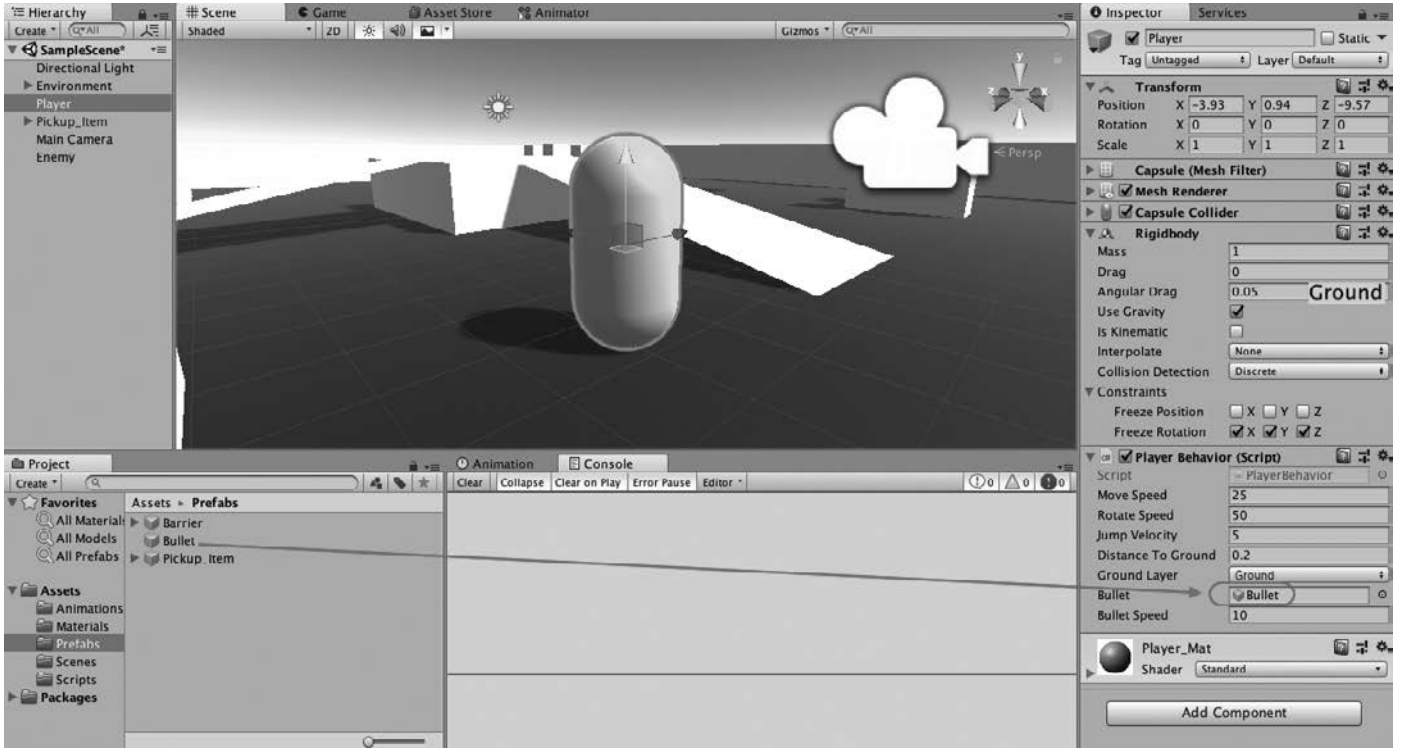

Hierarchy
Create
SampleScene*
Directional Light
Environment
Player
Pickup_Item
Main Camera
Enemy
Scene
Game
Asset Store
Animator
Shaded
2D
Gizmos
Persp
Inspector
Services
Player
Static
Tag
Untagged
Layer
Default
Transform
Position
Rotation
Scale
Capsule (Mesh Filter)
Mesh Renderer
Capsule Collider
Rigidbody
Mass
Drag
Angular Drag
Ground
Use Gravity
Is Kinematic
Interpolate
None
Collision Detection
Discrete
Constraints
Freeze Position
Freeze Rotation
Player Behavior (Script)
Script
PlayerBehavior
Move Speed
25
Rotate Speed
50
Jump Velocity
5
Distance To Ground
0.2
Ground Layer
Ground
Bullet
Bullet Speed
10
Player_Mat
Shader
Standard
Add Component
Project
Create
Favorites
All Materials
All Models
All Prefabs
Assets
Animations
Materials
Prefabs
Scenes
Scripts
Packages
Assets ▸ Prefabs
Barrier
Bullet
Pickup_Item
Animation
Console
Clear
Collapse
Clear on Play
Error Pause
Editor

Рис. 8.6

3. Мы создаем локальную переменную `GameObject` каждый раз, когда нажимаем левую кнопку мыши:
   - используем метод `Instantiate()`, чтобы назначить свойству `GameObject` значение `newBullet`, передав префаб `Bullet`. Также используем положение капсулы, чтобы пуля появилась перед игроком, тем самым избегая столкновений;
   - в конце добавляем строчку `as GameObject`, чтобы явно привести возвращаемый объект к тому же типу, что и `newBullet`.
4. Вызываем метод `GetComponent()`, чтобы вернуть и сохранить компонент `Rigidbody` объекта `newBullet`.
5. Устанавливаем свойство `velocity` компонента `Rigidbody` в направлении `transform.forward` игрока и умножаем его на `bulletSpeed`:
   - изменение параметра `velocity` вместо использования `AddForce()` гарантирует, что гравитация не будет тянуть пули к земле по дуге при выстреле.

Опять же мы существенно обновили логику в сценарии игрока. Однако проблема в том, что ваша сцена и иерархия заполоняются объектами (снарядами). Наша следующая задача — удалять объекты спустя какое-то время после выстрела, чтобы избежать проблем с производительностью.

Следим за размножением GameObject

Независимо от того, пишем мы чисто кодовое приложение или трехмерную игру, важно регулярно удалять неиспользуемые объекты, чтобы не перегружать программу. После выстрела снаряды нам уже не нужны, но они все равно будут лежать на полу возле любой стены или объекта, с которым столкнется снаряд.

При реализации стрельбы на сцене могут появиться сотни, если не тысячи, пуль, которые нам уже не нужны. Ваша следующая задача — уничтожать каждую пулю по происшествии некоторого времени.

Время действовать. Уничтожаем пули

Решить эту задачу мы можем с помощью уже имеющихся знаний и сделать так, чтобы пули сами уничтожались в какой-то момент.

1. Создайте новый сценарий C# в папке Scripts и назовите его `BulletBehavior`.
2. Перетащите сценарий `BulletBehavior` на префаб `Bullet` в папке Prefabs и добавьте следующий код:

```
public class BulletBehavior : MonoBehaviour
{
    // 1
    public float onscreenDelay = 3f;

    void Start ()
    {
        // 2
        Destroy(this.gameObject, onscreenDelay);
    }
}
```

Разберем этот код.

- Мы объявляем переменную типа `float`, хранящую время, в течение которого префабы `Bullet` должны существовать на сцене после того, как будут созданы.
- Удаляем `GameObject` мы с помощью метода `Destroy()`.
  - Методу `Destroy()` в качестве параметра всегда нужен объект. В данном случае мы используем ключевое слово `this`, ссылаясь на объект, к которому прикреплен сценарий.
  - Метод `Destroy()` может принимать дополнительный параметр типа `float` — время задержки, после которого пуля будет уничтожаться.

Запустите игру, выстрелите несколько раз и посмотрите, как по истечении времени снаряды сами будут пропадать из иерархии объектов. Это значит, что пуля сама отрабатывает свое поведение, и никакие другие сценарии

не указывают ей, что делать. Это идеальный пример применения паттерна проектирования `Component`. Мы поговорим о нем подробнее в главе 12.

Теперь, покончив с уборкой, поговорим о ключевом компоненте любого хорошо спроектированного и организованного проекта — классе менеджера.

Создание игрового менеджера

Распространенное заблуждение, часто возникающее при обучении программированию, состоит в том, что все переменные должны быть публичными. Но в целом это не очень хорошая идея. По моему опыту, переменные по умолчанию должны быть приватными, и делать их публичными нужно только в случае необходимости. Вы еще убедитесь, что опытные программисты защищают свои данные с помощью классов менеджеров, и поскольку мы хотим выработать хорошие привычки, то будем следовать их примеру. Классы менеджеров — это своего рода воронки, которые позволяют безопасно получать доступ к важным переменным и методам.

Когда я говорю «безопасно», я именно это и имею в виду, что может показаться непривычным в контексте программирования. Однако когда в проекте есть разные классы, которые взаимно обмениваются данными и меняют состояния друг друга, проект быстро превращается в хаос. Вот почему наличие единой точки контакта, а именно менеджера, позволяет упорядочить игру. В следующем подразделе мы узнаем, как сделать это эффективно.

Отслеживание свойств игрока

Hero Born — простая игра, и в ней нам нужно отслеживать лишь две точки данных: сколько предметов собрал игрок и сколько у него осталось здоровья. Мы хотим, чтобы эти переменные были приватными и чтобы их мог изменять только класс менеджера. За счет этого мы получаем контроль над проектом и обеспечиваем безопасность данных. Ваша следующая задача — создать игровой менеджер для игры Hero Born и наполнить его полезными функциями.

Время действовать. Создаем игровой менеджер

Классы игрового менеджера будут использоваться в любых проектах, которые нам предстоит разрабатывать, поэтому узнаем, как правильно его создать.

1. Создайте новый сценарий C# в папке Scripts и назовите его `GameBehavior`.

Обычно данный сценарий называется GameManager, но в Unity это имя зарезервировано для внутренних целей. Если вы создаете сценарий и рядом с его именем вместо значка файла C# отображается значок шестеренки, то это означает, что работа сценария ограничена.

2. Создайте новый пустой игровой объект с помощью команды Create ▸ Create Empty и назовите его `GameManager`.
3. Прикрепите сценарий `GameBehavior.cs` к объекту `GameManager`, как показано на рис. 8.7.

Сценарии менеджера и другие файлы, не относящиеся к игре, настраиваются на пустые объекты, которые размещаются на сцене и не взаимодействуют с реальным трехмерным пространством.

4. Добавьте в сценарий `GameBehavior` следующий код:

```
public class GameBehavior : MonoBehaviour
{
    private int _itemsCollected = 0;
    private int _playerHP = 10;
}
```

Разберем этот код.

Мы добавили две новые приватные переменные, в которых хранится информация о количестве поднятых предметов и оставшемся здоровье. Эти переменные являются приватными, поскольку их можно изменять только в этом классе. Будь они публичными, другие классы могли бы изменять эти параметры, в результате чего в переменной могли бы оказаться неверные данные.

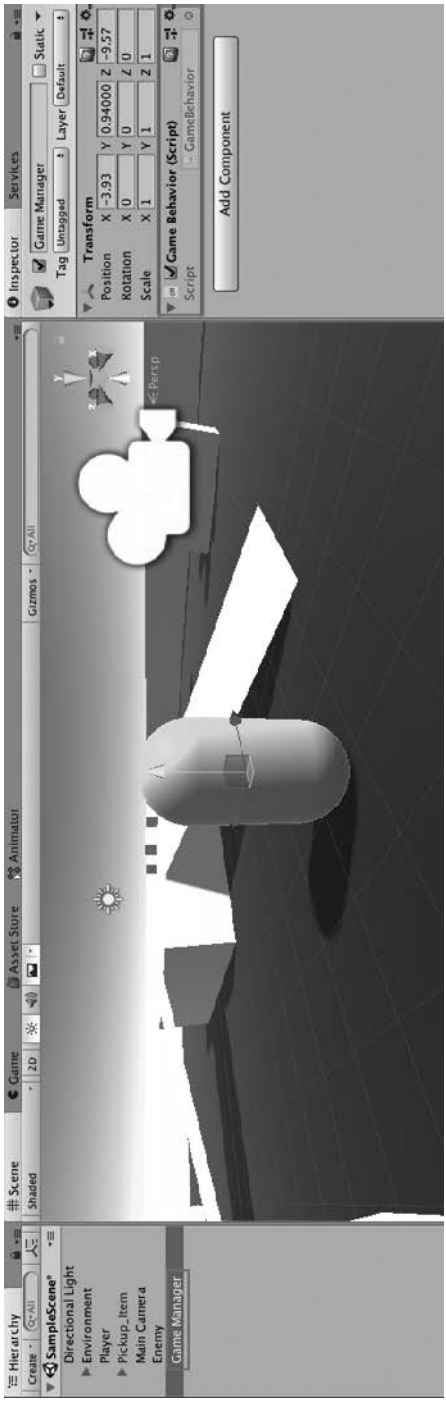

Hierarchy
Create
SampleScene*
Directional Light
Environment
Player
Pickup_Item
Main Camera
Enemy
Game Manager
Scene
Shaded
2D
Game
Asset Store
Animator
Gizmos
Persp
Inspector
Services
Game Manager
Static
Tag
Untagged
Layer
Default
Transform
Position
Rotation
Scale
X -3.93
Y 0.94000
Z -9.57
X 0
Y 0
Z 0
X 1
Y 1
Z 1
Game Behavior (Script)
Script
GameBehavior
Add Component

Рис. 8.7

Объявление переменных как приватных означает, что вы сами определяете, кто и как имеет доступ к ним. В следующем подразделе, посвященном свойствам для чтения (`get`) и для записи (`set`), мы познакомимся со стандартным безопасным способом решения этой задачи в будущем.

Свойства для чтения и для записи

Мы настроили сценарий менеджера и приватные переменные, но как получить к ним доступ из других классов, если они приватные? Мы могли бы написать отдельные публичные методы в сценарии `GameBehavior`, в которых обрабатывали бы передачу новых значений в публичные переменные. Но, возможно, есть способ получше.

На такой случай C# предоставляет всем переменным свойства для чтения (`get`) и для записи (`set`), идеально подходящие для нашей задачи. Это методы, которые автоматически запускаются компилятором C# независимо от того, вызываем мы их явно или нет, подобно методам `Start()` и `Update()`, которые выполняются в Unity при запуске сцены.

Свойства для чтения и для записи могут быть добавлены к любой переменной с начальным значением или без него, как показано в следующем фрагменте кода:

```
public string firstName { get; set; };

OR

public string lastName { get; set; } = "Smith";
```

Однако их использование не дает никаких дополнительных преимуществ. Чтобы все заработало должным образом, необходимо добавить блок кода для каждого свойства, как показано ниже:

```
public string FirstName
{
    get {
        // Строки кода выполняются при доступе к переменной
    }

    set {
        // Строки кода выполняются при обновлении переменной
    }
}
```

Теперь у нас есть свойства для чтения и для записи, выполняющие дополнительные действия, которые нам нужны в определенной ситуации.

Но это еще не все, поскольку нам все еще необходимо обработать эту новую логику.

Каждый блок кода `get` должен возвращать значение, а каждый блок `set` — присваивать. Здесь нам потребуется комбинация приватной переменной, которую можно назвать поддерживающей, и публичной переменной со свойствами для чтения и для записи. Приватная переменная остается защищенной, а публичная позволяет обращаться к ней из других классов, как показано в следующем фрагменте кода:

```
private string _firstName
public string FirstName {
    get {
        return _firstName;
    }

    set {
        _firstName = value;
    }
}
```

Разберем этот код.

- Мы можем вернуть значение, хранящееся в приватной переменной, с помощью `get` в любой момент, когда это значение будет нужно другому классу, не предоставляя при этом прямой доступ данному классу.
- Мы можем изменить значение приватной переменной в любой момент, когда внешний класс присваивает новое значение публичной переменной, таким образом реализуя их синхронизацию.
- За ключевым словом `value` скрывается переданное для установки значение.

Описанная методика может показаться несколько экзотической, когда отделена от реального приложения, поэтому добавим в сценарий `GameBehavior` публичные переменные со свойствами для чтения и для записи, чтобы они соответствовали уже созданным приватным переменным.

Время действовать. Добавляем вспомогательные переменные

Теперь, освоив синтаксис доступа к свойствам для чтения и для записи, мы можем реализовать оба свойства в нашем классе менеджера, чтобы добиться большей эффективности и читабельности кода.

Добавьте в сценарий GameBehavior вот этот код:

```
public class GameBehavior : MonoBehaviour
{
    private int _itemsCollected = 0;

    // 1
     public int Items
     {
         // 2
         get { return _itemsCollected; }

         // 3
         set {
                _itemsCollected = value;
                Debug.LogFormat("Items: {0}", _itemsCollected);
         }
      }

     private int _playerHP = 10;

     // 4
     public int HP
     {
         get { return _playerHP; }
         set {
                _playerHP = value;
                Debug.LogFormat("Lives: {0}", _playerHP);
          }
     }
}
```

Разберем этот код.

- Мы объявили новую публичную переменную Items со свойствами для чтения и для записи.
- С помощью get вернем значение, хранящееся в _itemsCollected, когда внешний класс пытается обратиться к Items.
- С помощью set мы присваиваем _itemsCollected новое значение Items при каждом обновлении, добавив сюда также вызов функции Debug.LogFormat(), чтобы вывести измененное значение _itemsCollected.
- Мы создали публичную переменную с именем HP и свойствами для чтения и для записи, которую будем использовать в комбинации с приватной переменной _playerHP.

Обе приватные переменные теперь доступны для чтения, но только через их публичные аналоги, которые можно изменить лишь в `GameBehavior`. Благодаря такой методике мы гарантируем, что приватные данные могут быть доступны и изменены только из определенных мест. Это упрощает взаимодействие с `GameBehavior` из других реализующих механику сценариев, а также позволяет отображать данные в реальном времени в простом пользовательском интерфейсе, который мы создадим в конце главы.

Проверим, как это работает, обновив свойство `Items` после успешного поднятия бонуса на арене.

Время действовать. Обновляем коллекцию предметов

Теперь, настроив в сценарии `GameBehavior` все переменные, мы можем обновлять `Items` каждый раз, когда подбираем бонус.

1. Добавьте следующий код в сценарий `ItemBehavior`:

```
public class ItemBehavior : MonoBehaviour
 {
     // 1
     public GameBehavior gameManager;

     void Start()
     {
           // 2
           gameManager = GameObject.Find("Game
             Manager").GetComponent<GameBehavior>();
     }

     void OnCollisionEnter(Collision collision)
     {
         if (collision.gameObject.name == "Player")
         {
             Destroy(this.transform.parent.gameObject);
             Debug.Log("Item collected!");

             // 3
             gameManager.Items += 1;
         }
     }
}
```

2. Нажмите кнопку Play и подберите бонус. После этого вы увидите в консоли сообщение, выведенное из сценария менеджера, как показано на рис. 8.8.

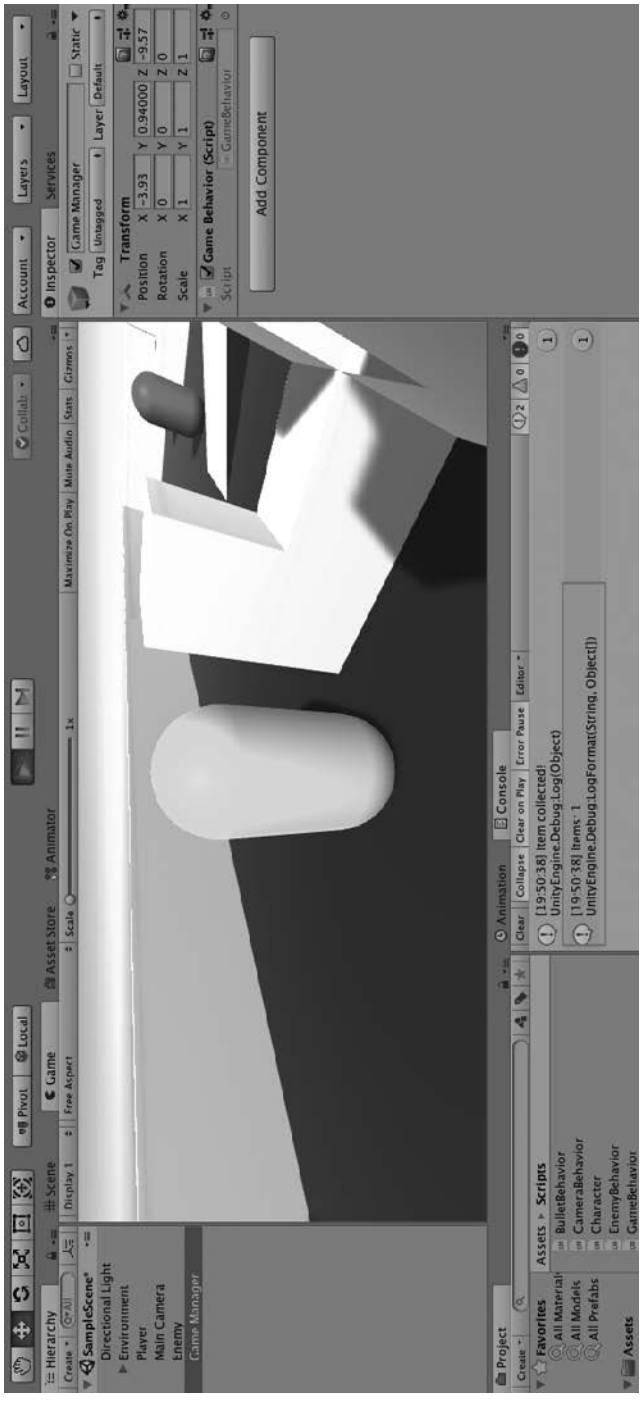

Hierarchy
Create
SampleScene*
Directional Light
Environment
Player
Main Camera
Enemy
Game Manager
Scene
Game
Asset Store
Animator
Pivot
Local
Collab
Account
Layers
Layout
Inspector
Services
Game Manager
Static
Tag Untagged
Layer Default
Transform
Position X -3.93 Y 0.94000 Z -9.57
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Game Behavior (Script)
Script GameBehavior
Add Component
Project
Favorites
All Materials
All Models
All Prefabs
Assets
Scripts
BulletBehavior
CameraBehavior
Character
EnemyBehavior
GameBehavior
Animation
Console
Clear
Collapse
Clear on Play
Error Pause
Editor
[19:50:38] Item collected!
UnityEngine.Debug:Log(Object)
[19:50:38] Items: 1
UnityEngine.Debug:LogFormat(String, Object[])

Рис. 8.8

Разберем этот код.

- Мы создаем новую переменную типа `GameBehavior` и храним в ней ссылку на прикрепленный сценарий.
- В методе `Start()` мы инициализируем `gameManager`, осмотрев сцену методом `Find()`, а затем вызываем функцию `GetComponent()`.

Подобный код довольно часто пишется в одной строке как в документации Unity, так и в проектах, созданных сообществом. Это сделано для простоты, но если вам удобнее писать метод Find() и GetComponent() отдельно, то так и делайте. Нет ничего плохого в ясном и понятном форматировании.

- Мы увеличиваем свойство `Items` в классе `gameManager` в методе `OnCollisionEnter()` после уничтожения префаба `Item`.

Поскольку мы уже настроили сценарий `ItemBehavior` на обработку столкновений, нам не составляет труда добавить в метод `OnCollisionEnter()` связь с классом менеджера, когда игрок берет элемент. Следует понимать, что подобное разделение функций делает код более гибким и в результате он с меньшей вероятностью выйдет из строя при внесении изменений во время разработки.

Последнее, чего не хватает игре Hero Born, — какого-нибудь интерфейса, где отображались бы нужные для игрока данные. В программировании и разработке игр это называется пользовательским интерфейсом (user interface, UI). Ваша последняя задача в данной главе — ознакомиться с тем, как Unity создает и обрабатывает код UI.

Улучшаем игрока

На данный момент мы написали несколько сценариев, которые совместными усилиями позволяют игроку наслаждаться механиками передвижения, прыжка, подбора предметов и стрельбы. Однако нам по-прежнему не хватает какого-либо дисплея или визуальной подсказки, где отображались бы данные нашего игрока, а также условия поражения или победы. Именно на этих двух темах мы сосредоточимся в последнем разделе.

Графический интерфейс

Пользовательский интерфейс — видимая часть любой компьютерной системы. Указатель мыши, значки папок и программы на вашем ноутбуке — все это элементы UI. Для нашей игры нам нужен простой дисплей, чтобы игрок мог понимать, сколько предметов он собрал и сколько у него здоровья. Кроме того, нам потребуется текстовое поле, в котором будет выводиться информация об определенных событиях.

Элементы пользовательского интерфейса в Unity можно добавлять двумя способами:

- непосредственно из меню Create на панели Hierarchy, как и в случае с любым другим `GameObject`;
- с помощью встроенного класса `GUI` в коде.

Нам для данного проекта больше подойдет версия с кодом. Мы добавим три элемента пользовательского интерфейса в класс `GameBehavior`. Это не означает, что один подход лучше, чем другой, но мы учимся именно программировать, поэтому будем придерживаться вариантов, связанных с кодом.

У класса `GUI` есть несколько методов для создания и размещения компонентов. Все вызовы элементов графического интерфейса выполняются в методе `MonoBehaviour`, называемом `OnGUI()`. Метод `OnGUI()` работает подобно методу `Update()` для всего пользовательского интерфейса. Он выполняется от одного до нескольких раз за кадр и является основной точкой программной привязки интерфейсов.

В следующем примере мы затронем лишь верхушку айсберга класса GUI, но, поскольку его возможности невероятно обширны, нельзя просто так взять и пройти мимо документации, не заглянув в нее: docs.unity3d.com/ScriptReference/GUI.html.

Если вам интересно узнать о непрограммном создании интерфейса, то посмотрите серию видеоуроков по Unity: unity3d.com/learn/tutorials/s/userinterface-ui.

Ваша следующая задача — добавить на игровую сцену простой пользовательский интерфейс для отображения количества собранных предметов и здоровья игрока, хранящихся в сценарии `GameBehavior`.

Время действовать. Добавляем элементы пользовательского интерфейса

На данный момент у нас не так много всего, что стоило бы показывать игроку, но имеющееся должно отображаться на экране в приятной и заметной форме. Выполните следующие действия.

1. Добавьте в сценарий `GameBehavior` следующий код для подбора предмета:

```
public class GameBehavior : MonoBehaviour
{
    // 1
    public string labelText = "Collect all 4 items and win your
    freedom!";
    public int maxItems = 4;

    private int _itemsCollected = 0;
    public int Items
    {
        get { return _itemsCollected; }
        set {
            _itemsCollected = value;

            // 2
            if(_itemsCollected >= maxItems)
            {
                labelText = "You've found all the items!";
            }
            else
            {
                labelText = "Item found, only " +
                  (maxItems - _itemsCollected) + " more to go!";
            }
        }
    }

    private int _playerHP = 3;
    public int HP
    {
        get { return _playerHP; }
        set {
            _playerHP = value;
            Debug.LogFormat("Lives: {0}", _playerHP);
        }
    }
```

```
    // 3
    void OnGUI()
    {
        // 4
        GUI.Box(new Rect(20, 20, 150, 25),
          "Player Health:" + _playerHP);

        // 5
        GUI.Box(new Rect(20, 50, 150, 25),
          "Items Collected: " + _itemsCollected);

        // 6
        GUI.Label(new Rect(Screen.width / 2 - 100, Screen.height - 50,
          300, 50), labelText);
    }
}
```

2. Запустите игру и посмотрите на появившиеся элементы интерфейса, показанные на рис. 8.9.

Разберем этот код.

- Мы создаем две новые публичные переменные:
  - в одной мы будем хранить текст, который выводится в нижней части экрана;
  - в другой мы храним максимальное количество предметов на уровне.
- Объявляем оператор `if` в свойстве для записи объекта `_itemsCollected`:
  - если игрок собрал больше или равно `maxItems`, то он выиграл, о чем нам сообщит `labelText`;
  - в противном случае `labelText` показывает, сколько предметов осталось собрать.
- Объявляем метод `OnGUI()`, в котором разместим код пользовательского интерфейса.
- Создаем объект `GUI.Box()` с указанными размерами и строковым сообщением:
  - конструктор класса `Rect` принимает значения `x`, `y` — ширину и высоту;
  - позиция `Rect` отсчитывается от верхнего левого угла экрана;

Hierarchy
Create
SampleScene
Directional Light
Environment
Player
Pickup_Item
Main Camera
Enemy
Game Manager
Scene
Game
Asset Store
Animator
Pivot
Local
Collab
Account
Layers
Layout
Display 1
Free Aspect
Scale
1x
Maximize On Play
Mute Audio
Stats
Gizmos
Player Health: 3
Items Collected: 0
Collect all 4 items and win, our freedom!
Inspector
Services
Player
Static
Tag
Untagged
Layer
Default
Transform
Position
Rotation
Scale
Capsule (Mesh Filter)
Mesh Renderer
Capsule Collider
Rigidbody
Mass
Drag
Angular Drag
Use Gravity
Is Kinematic
Interpolate
Collision Detection
Constraints
None
Discrete

Рис. 8.9

    - конструктор `new Rect (20, 20, 150, 25)` преобразуется в 2D-блок в верхнем левом углу, который находится в 20 единицах от левого края, в 20 единицах от верхнего, имеет ширину 150 и высоту 25.
- Мы создаем еще один `GUI.Box()` под полем здоровья, чтобы отображать текущее количество элементов.
- Мы создаем `GUI.Label()` по центру внизу экрана для отображения `labelText`:
    - поскольку метод `OnGUI()` запускается не менее раза за кадр, всякий раз при изменении значения `labelText` изменения на экране будут видны мгновенно;
    - вместо того чтобы вручную с линейкой отмерять середину экрана, мы используем свойства ширины и высоты класса `Screen`, чтобы получить абсолютные значения.

Теперь, запустив игру, мы увидим три новых элемента пользовательского интерфейса с соответствующими значениями. При подборе предмета обновляются счетчики `labelText` и `_itemsCollected`, как показано на рис. 8.10.

В любой игре можно выиграть или проиграть. Далее мы реализуем условия победы и поражения и соответствующий им пользовательский интерфейс.

Условия победы и поражения

Мы реализовали базовую игровую механику и простой пользовательский интерфейс, но в Hero Born все еще отсутствует важный элемент игрового дизайна: условия победы и поражения. Они определяют, когда игрок выигрывает или проигрывает в игре, и выполняют тот или иной код в зависимости от ситуации.

В наброске игрового документа в главе 6 мы изложили условия победы и поражения следующим образом:

- чтобы победить, нужно собрать на уровне все предметы и сохранить хотя бы 1 очко здоровья;
- чтобы проиграть, нужно получать урон от врагов до тех пор, пока количество очков здоровья не станет равным 0.

Количество предметов обновлено
Информация обновлена
Player Health: 3
Items Collected: 1
Item found, only 3 more to go!

Рис. 8.10

Эти условия будут воздействовать и на UI, и на игровую механику, но мы уже настроили сценарий GameBehavior, чтобы данный вопрос был решен. Наши свойства для чтения и для записи станут обрабатывать любую логику, связанную с игрой, а метод OnGUI() — производить изменения пользовательского интерфейса, когда игрок выигрывает или проигрывает.

В этом подразделе мы собираемся реализовать условие победы, поскольку подбирать предметы наш игрок уже умеет. Когда мы перейдем к поведению вражеского ИИ в главе 9, добавим логику условия поражения. Ваша следующая задача — определить с помощью кода, победил ли игрок.

Время действовать. Обозначаем победу в игре

Нам всегда хочется дать игроку четкую и мгновенную обратную связь, поэтому начнем с добавления условия победы, как показано ниже.

1. Добавьте в сценарий GameBehavior следующий код:

```
public class GameBehavior : MonoBehaviour
{
    // 1
    public bool showWinScreen = false;

    private int _itemsCollected = 0;
    public int Items
    {
        get { return _itemsCollected; }
        set {
            _itemsCollected = value;

            if (_itemsCollected >= maxItems)
            {
                labelText = "You've found all the items!";

                // 2
                showWinScreen = true;
            }
            else
            {
                labelText = "Item found, only " +
                  (maxItems - _itemsCollected) + " more to go!";
            }
        }
    }
```

```
// ... Изменения не требуются ...

    void OnGUI()
    {
        // ... В GUI изменения не требуются ...

        // 3
        if (showWinScreen)
        {
            // 4
            if (GUI.Button(new Rect(Screen.width/2 - 100,
                Screen.height/2 - 50, 200, 100), "YOU WON!"))
            {

            }
        }
    }
}
```

2. Задайте переменной `Max Items` значение `1` на панели Inspector, чтобы проверить экран победы, как показано на рис. 8.11.

Разберем этот код.

- Мы создаем новую переменную типа `bool`, чтобы отслеживать, когда должен появиться экран победы.
- Устанавливаем для `showWinScreen` значение `true` в свойстве для записи объекта `Items`, когда игрок собрал все предметы.
- Используем оператор `if` внутри метода `OnGUI()`, чтобы проверить, должен ли отображаться экран победы.
- Создаем интерактивный объект `GUI.Button()` в центре экрана с сообщением для игрока:
  - метод `GUI.Button()` возвращает значение `bool` — `true`, когда кнопка нажата, и `false`, когда нет.
  - встраивание вызова `GUI.Button()` внутри оператора `if` выполняет тело оператора `if` при нажатии кнопки.

Если для параметра `Max Items` установлено значение `1`, кнопка выигрыша появится при сборе единственного имеющегося `Pickup_Item` в сцене. Нажатие кнопки сейчас ни к чему не ведет, но мы поправим это в следующем разделе.

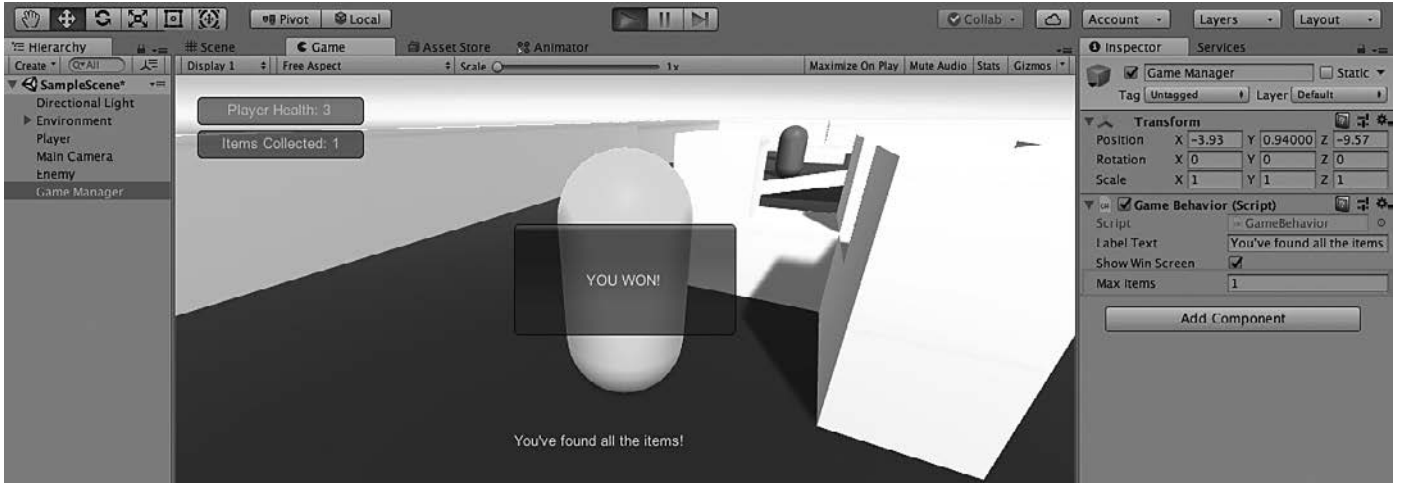

Pivot
Local
Collab
Account
Layers
Layout
Hierarchy
Scene
Game
Asset Store
Animator
Inspector
Services
Create
Display 1
Free Aspect
Scale
Maximize On Play
Mute Audio
Stats
Gizmos
SampleScene*
Directional Light
Environment
Player
Main Camera
Enemy
Game Manager
Player Health: 3
Items Collected: 1
YOU WON!
You've found all the items!
Game Manager
Static
Tag
Untagged
Layer
Default
Transform
Position
Rotation
Scale
Game Behavior (Script)
Script
GameBehavior
Label Text
You've found all the items
Show Win Screen
Max Items
Add Component

Рис. 8.11

Использование директив и пространств имен

В данный момент условие победы работает так, как ожидалось, но игрок даже после победы все еще может управлять капсулой и не имеет возможности перезапустить игру после ее завершения. У Unity в классе `Time` есть свойство `timeScale`, которое при значении `0` замораживает игровую сцену. Однако для перезапуска игры нам нужен доступ к пространству имен `SceneManagement`, которое по умолчанию недоступно из наших классов.

Пространство имен — это сгруппированный под определенным именем набор классов, служащий для организации больших проектов и предотвращения конфликтов между сценариями, которые могут иметь одинаковые имена. Для получения доступа к классам пространства имен необходимо добавить в класс директиву `using`.

Все сценарии C#, созданные из Unity, изначально начинаются с трех элементов `using` по умолчанию, показанных ниже:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

Они позволяют получить доступ к общим пространствам имен, но в Unity и C# есть много другого, что можно добавить с помощью ключевого слова `using`, за которым следует имя пространства имен.

Время действовать. Настраиваем паузу и рестарт

Поскольку игра должна ставиться на паузу, когда игрок выигрывает или проигрывает, самое время обратиться к пространству имен, которое по умолчанию не включено в новые сценарии C#.

Добавьте следующий код в сценарий `GameBehavior` и запустите игру:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// 1
using UnityEngine.SceneManagement;
```

```
public class GameBehavior : MonoBehaviour
{
    // ... Изменения не требуются ...

    private int _itemsCollected = 0;
    public int Items
    {
        get { return _itemsCollected; }
        set {
            _itemsCollected = value;

            if (_itemsCollected >= maxItems)
            {
                labelText = "You've found all the items!";
                showWinScreen = true;

                // 2
                Time.timeScale = 0f;
            }
            else
            {
                labelText = "Item found, only " + (maxItems -
                _itemsCollected) + " more to go!";
            }
        }
    }

    // ... Другие изменения не требуются ...

    void OnGUI()
    {
        // ... В GUI изменения не требуются ...

        if (showWinScreen)
        {
            if (GUI.Button(new Rect(Screen.width/2 - 100,
                Screen.height/2 - 50, 200, 100), "YOU WON!"))
            {
                // 3
                SceneManager.LoadScene(0);

                // 4
                Time.timeScale = 1.0f;
            }
        }
    }
}
```

Разберем этот код.

- С помощью ключевого слова `using` мы добавляем пространство имен `SceneManagement`, которое обрабатывает всю логику, связанную со сценой в Unity.
- Мы устанавливаем для свойства `Time.timeScale` значение `0`, чтобы приостанавливать игру, когда отображается экран победы. Любой ввод со стороны пользователя будет игнорироваться.
- Мы вызываем метод `LoadScene()` при нажатии кнопки экрана выигрыша:
  - метод `LoadScene()` принимает индекс сцены как параметр `int`;
  - поскольку в нашем проекте есть только одна сцена, мы используем индекс `0`, чтобы перезапустить игру с самого начала.
- Мы сбрасываем свойство `Time.timeScale` на значение по умолчанию `1`, чтобы при перезапуске сцены все элементы управления ожили.

Теперь, когда вы подбираете предмет и нажимаете кнопку на экране победы, уровень перезапускается, все сценарии и компоненты восстанавливаются до своих исходных значений и настраиваются для следующего раунда!

Подведем итоги

Поздравляю! В Hero Born теперь можно играть. Мы реализовали механику прыжков и стрельбы, обработали физические столкновения и создание объектов, а также добавили несколько основных элементов пользовательского интерфейса для отображения обратной связи. Мы даже дошли до перезапуска уровня при победе игрока.

В данной главе мы рассмотрели много новых тем, и не забывайте возвращаться, повторять материал, чтобы закрепить понимание того, из чего состоит код, который мы написали. Обратите особое внимание на перечисления, свойства для записи и для чтения, а также пространство имен. С этого момента по мере вашего узнавания возможностей языка C# код будет только усложняться.

В следующей главе мы научим вражеские объекты `GameObject` обращать внимание на нашего игрока, когда мы будем подбираться слишком близко. После обнаружения враг начнет преследовать нас и стрелять, что усложнит жизнь для нашего игрока.

Контрольные вопросы. Работа с механикой

1. Какие типы данных хранятся в перечислениях?
2. Как создать копию префаба `GameObject` в запущенной сцене?
3. Какие свойства переменных позволяют добавлять функциональные возможности, когда к их значению обращаются или пытаются его изменить?
4. Какой метод Unity отображает все объекты пользовательского интерфейса в сцене?

9 Основы ИИ и поведение врагов

Чтобы реализуемые в игре механики казались реальными, игрок должен преодолевать сложности, ощущать их последствия и получать награды. Без этих трех вещей игроку незачем переживать за своего персонажа, не говоря уже о продолжении игры. И хотя существует множество игровых механик, отвечающих одному или нескольким из этих условий, ничто не сравнится с противником, который пытается найти и уничтожить вас.

Создать умного врага — непростая задача, и часто она чревата долгими стараниями и разочарованиями. Однако Unity имеет встроенные функции, компоненты и классы, которые можно задействовать для проектирования и реализации систем ИИ в более удобной для пользователя форме.

Эти инструменты помогут нам закончить Hero Born и создать фундамент для изучения более сложных тем языка C#.

В этой главе мы рассмотрим такие темы, как:

- система навигации Unity;
- статические объекты и навигационные сетки;
- навигационные агенты;
- процедурное программирование и логика;
- получение и нанесение урона;
- условие поражения;
- методики рефакторинга.

Навигация с помощью Unity

Когда мы говорим о навигации в реальной жизни, то обычно имеем в виду вопрос о том, как добраться из точки А в точку Б. Навигация в виртуальном трехмерном пространстве работает примерно так же, но как учесть опыт, который люди успели накопить с момента нашего появления? Мы ведь не сразу всему научились. Все навыки, от ходьбы по плоской поверхности до подъема по лестнице и прыжков с бордюров, мы приобрели путем практики. Как запрограммировать все это в игру и не сойти с ума?

Прежде чем вы сможете ответить на любой из этих вопросов, вам нужно знать, какие компоненты навигации есть в Unity.

Компоненты навигации

Расскажу кратко. Команда Unity потратила много времени на совершенствование системы навигации и создание компонентов, которые мы можем использовать для управления перемещением игровых и неигровых персонажей. Каждый из рассмотренных далее компонентов входит в стандартную комплектацию Unity и имеет встроенные сложные функции.

- Компонент `NavMesh` — это, по сути, карта поверхностей, по которым на данном уровне можно ходить. Сам компонент создается из геометрии уровня в процессе, называемом запеканием. Добавление `NavMesh` на уровень позволяет создать уникальный ресурс, содержащий данные навигации.
- Если `NavMesh` — карта уровня, то `NavMeshAgent` — словно подвижная фигура на доске. Любой объект с присоединенным компонентом `NavMeshAgent` автоматически избегает других агентов или препятствий, с которыми вступает в контакт.
- Навигационная система должна знать о любых движущихся или неподвижных объектах на уровне, которые могут заставить `NavMeshAgent` изменить свой маршрут. Добавление компонентов `NavMeshObstacle` к этим объектам позволяет системе знать, что их следует избегать.

Это далеко не полное описание навигационной системы Unity, но и его достаточно, чтобы разобраться с поведением врага. В данной главе мы сосредоточимся на добавлении `NavMesh` на наш уровень, настройке префаба `Enemy` как `NavMeshAgent` и перемещении префаба `Enemy` для движения по заранее определенному маршруту.

В этой главе мы будем использовать только компоненты NavMesh и NavMeshAgent, но если вы хотите вдохнуть в уровень больше жизни, то почитайте, как создавать препятствия: docs.unity3d.com/ru/current/Manual/nav-CreateNavMeshObstacle.html.

Ваша первая задача в настройке «умного» врага — создать сетку `NavMesh` над «пешеходными зонами» арены.

Время действовать. Настраиваем NavMesh

Настроим компонент `NavMesh` нашего уровня.

1. Выберите объект `Environment`, щелкните на стрелочке рядом со словом Static на панели Inspector и выберите вариант Navigation Static (рис. 9.1).

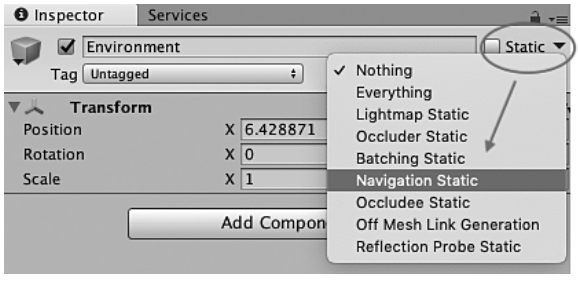


Рис. 9.1

2. Нажмите Yes, change children, когда появится диалоговое окно, чтобы установить все дочерние объекты `Environment` на объект `Navigation Static`.
3. Выберите пункт меню Windows ▸ AI ▸ Navigation и перейдите на вкладку Bake. Оставьте все значения по умолчанию и нажмите кнопку Bake (рис. 9.2).

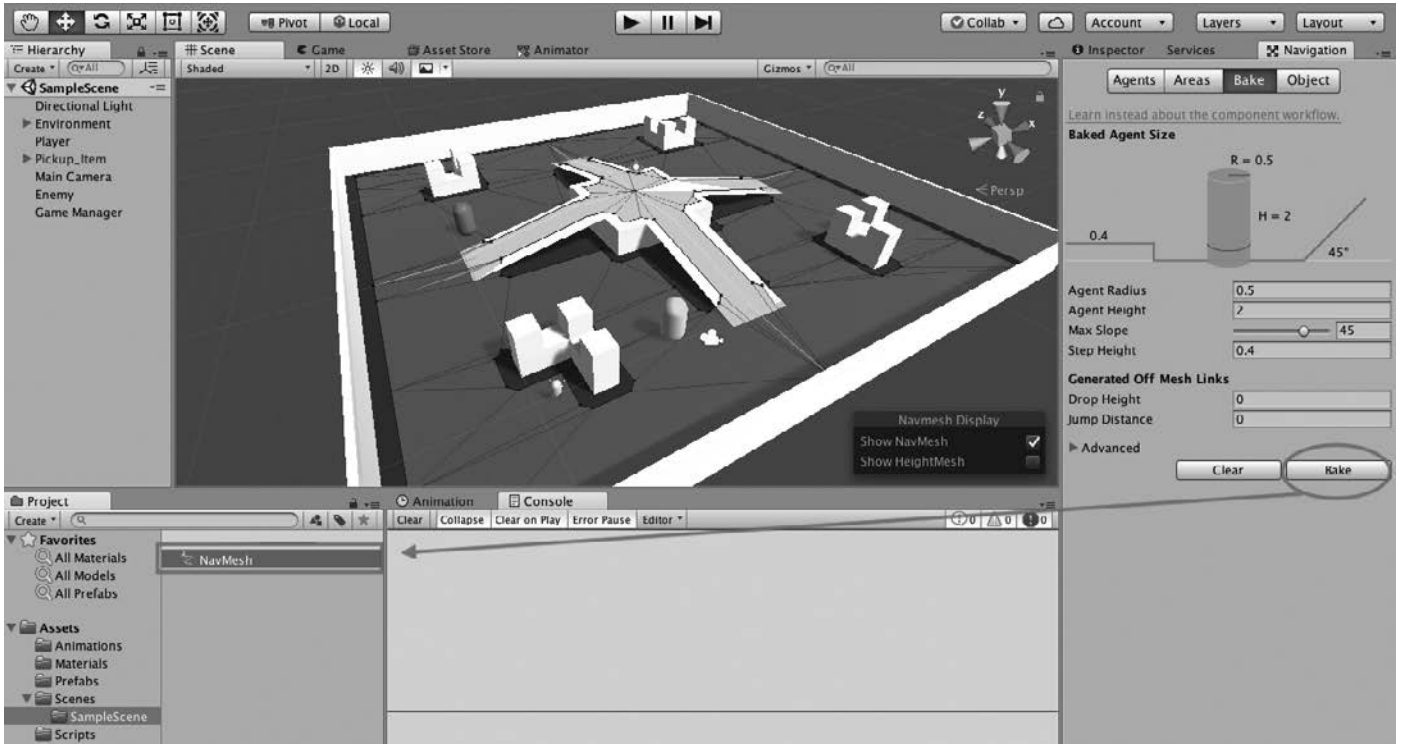

Pivot
Local
Collab
Account
Layers
Layout
Hierarchy
Scene
Game
Asset Store
Animator
Inspector
Services
Navigation
Create
Shaded
2D
Gizmos
SampleScene
Directional Light
Environment
Player
Pickup_Item
Main Camera
Enemy
Game Manager
Persp
Navmesh Display
Show NavMesh
Show HeightMesh
Agents
Areas
Bake
Object
Learn instead about the component workflow.
Baked Agent Size
R = 0.5
H = 2
0.4
45°
Agent Radius
0.5
Agent Height
2
Max Slope
45
Step Height
0.4
Generated Off Mesh Links
Drop Height
0
Jump Distance
0
Advanced
Clear
Bake
Project
Animation
Console
Clear
Collapse
Clear on Play
Error Pause
Editor
Favorites
All Materials
All Models
All Prefabs
NavMesh
Assets
Animations
Materials
Prefabs
Scenes
SampleScene
Scripts

Рис. 9.2

Теперь все объекты уровня помечены как Navigation Static; это значит, наша недавно созданная сетка NavMesh оценила их доступность на основе настроек своего NavMeshAgent по умолчанию. Голубым цветом на предыдущем рисунке[1] обозначены поверхности, по которым может ходить любой объект с присоединенным компонентом NavMeshAgent. Кстати, его еще надо присоединить.

Время действовать. Настраиваем вражеских агентов

Зарегистрируем префаб Enemy как NavMeshAgent.

1. Выберите префаб Enemy, нажмите кнопку Add Component на панели Inspector и найдите NavMesh Agent. Убедитесь, что префаб Enemy на сцене обновился (рис. 9.3).
2. Выберите команду Create ▸ Create Empty в окне Hierarchy и назовите новый объект Patrol Route.
   - Выберите Patrol Route, нажмите Create ▸ Create Empty, чтобы добавить дочерний GameObject, и назовите его Location 1. Расположите Location 1 в одном из углов уровня (рис. 9.4).
3. Создайте еще три пустых дочерних объекта внутри Patrol Route, назовите их Location 2, Location 3 и Location 4 соответственно и расположите в оставшихся углах уровня, чтобы получился квадрат (рис. 9.5).

Добавление компонента NavMeshAgent к объекту Enemy дает компоненту NavMesh указание зарегистрировать врага как объект, имеющий доступ к его функциям автономной навигации.

Создание четырех пустых игровых объектов в каждом углу уровня позволяет нам нарисовать маршрут, который враг должен будет патрулировать. Группировка точек маршрута в пустой родительский объект упрощает обращение к ним в коде и позволяет избежать беспорядка в иерархии. Осталось лишь написать код, который заставляет врага идти по маршруту, и этим мы займемся в следующем разделе.

---

[1] Напомню, что PDF-файл с цветными иллюстрациями можно скачать по ссылке static.packt-cdn.com/downloads/9781800207806_ColorImages.pdf.

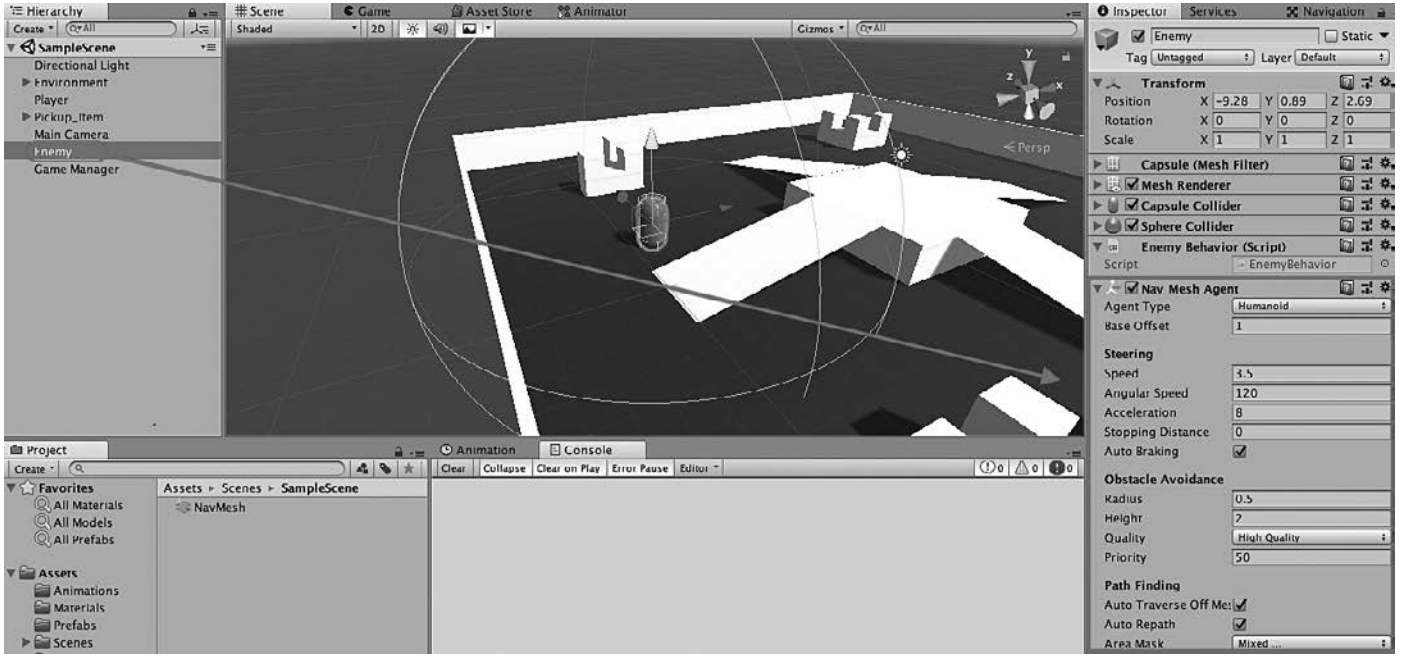

Hierarchy
Create
SampleScene
Directional Light
Environment
Player
Pickup_Item
Main Camera
Enemy
Game Manager
Scene
Game
Asset Store
Animator
Shaded
2D
Gizmos
Persp
Inspector
Services
Navigation
Enemy
Static
Tag
Untagged
Layer
Default
Transform
Position
Rotation
Scale
Capsule (Mesh Filter)
Mesh Renderer
Capsule Collider
Sphere Collider
Enemy Behavior (Script)
Script
EnemyBehavior
Nav Mesh Agent
Agent Type
Humanoid
Base Offset
Steering
Speed
Angular Speed
Acceleration
Stopping Distance
Auto Braking
Obstacle Avoidance
Radius
Height
Quality
High Quality
Priority
Path Finding
Auto Traverse Off Me
Auto Repath
Area Mask
Mixed ...
Project
Create
Favorites
All Materials
All Models
All Prefabs
Assets
Animations
Materials
Prefabs
Scenes
Assets ▸ Scenes ▸ SampleScene
NavMesh
Animation
Console
Clear
Collapse
Clear on Play
Error Pause
Editor

Рис. 9.3

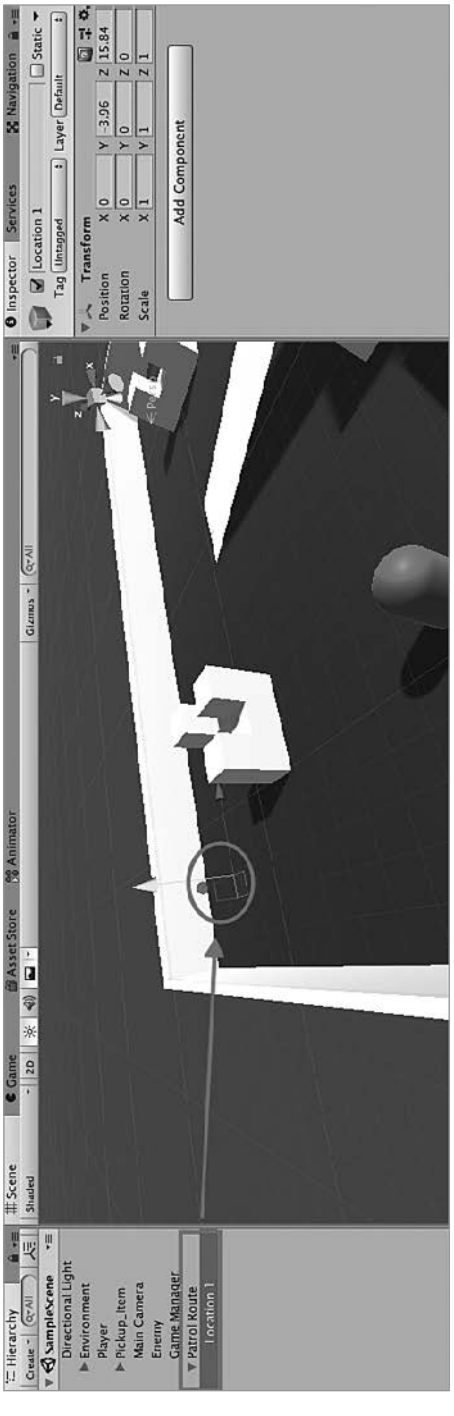

Hierarchy
Create
SampleScene
Directional Light
Environment
Player
Pickup_Item
Main Camera
Enemy
Game Manager
Patrol Route
Location 1
Scene
Shaded
2D
Game
Asset Store
Animator
Gizmos
Inspector
Services
Navigation
Location 1
Static
Tag
Untagged
Layer
Default
Transform
Position
Rotation
Scale
X 0
Y -3.96
Z 15.84
X 0
Y 0
Z 0
X 1
Y 1
Z 1
Add Component

Рис. 9.4

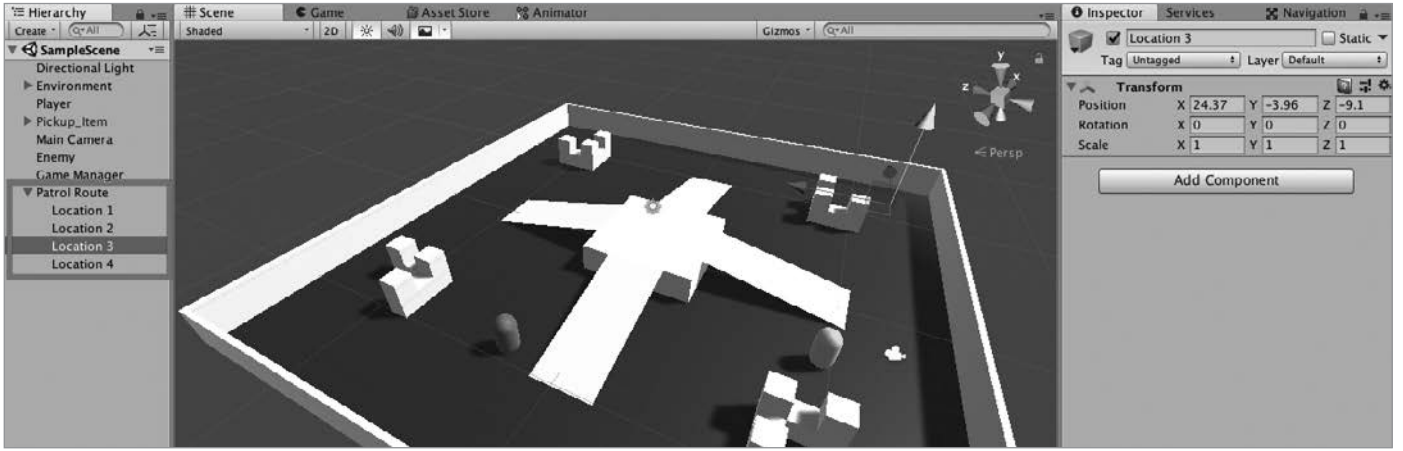

Hierarchy
Create
SampleScene
Directional Light
Environment
Player
Pickup_Item
Main Camera
Enemy
Game Manager
Patrol Route
Location 1
Location 2
Location 3
Location 4
Scene
Game
Asset Store
Animator
Shaded
2D
Gizmos
Persp
Inspector
Services
Navigation
Location 3
Static
Tag
Untagged
Layer
Default
Transform
Position
X 24.37
Y -3.96
Z -9.1
Rotation
X 0
Y 0
Z 0
Scale
X 1
Y 1
Z 1
Add Component

Рис. 9.5

Перемещение вражеских агентов

Точки для патрулирования установлены, а в префабе `Enemy` есть компонент `NavMeshAgent`, но теперь нам нужно выяснить, как ссылаться на эти места и заставить врага двигаться самостоятельно. Для этого нам сначала нужно поговорить о важной концепции в мире разработки программного обеспечения — о процедурном программировании.

Процедурное программирование

Несмотря на, казалось бы, говорящее название, идея процедурного программирования не так проста для понимания, пока вы не разберетесь с ней. Но, как только вы все поймете, проблем с процедурным кодом больше не будет.

Любая задача, которая выполняет одну и ту же логику на одном или нескольких последовательных объектах, отлично подходит для парадигмы процедурного программирования. Вы уже немного занимались процедурным программированием при отладке массивов, списков и словарей с помощью циклов `for` и `foreach`. Каждый раз при выполнении этих циклов вы выполняли один и тот же вызов метода `Debug.Log()`, последовательно перебирая каждый элемент. Теперь попробуем использовать данный навык, чтобы получить более полезный результат.

Одна из наиболее распространенных задач процедурного программирования — перемещение элементов из одной коллекции в другую с обработкой в процессе. Нам эта идея тоже подойдет, поскольку мы хотим ссылаться на каждый дочерний объект в пустом родительском объекте `patrolRoute` и сохранять их в списке.

Время действовать. Получение ссылок на точки патрулирования

Теперь, когда мы поняли основы процедурного программирования, пришло время попробовать получить точки маршрута патрулирования и собрать их в пригодный для использования список.

1. Добавьте следующий код в сценарий `EnemyBehaviour`:

```
public class EnemyBehavior : MonoBehaviour
 {
     // 1
     public Transform patrolRoute;

     // 2
     public List<Transform> locations;

     void Start()
     {
         // 3
         InitializePatrolRoute();
     }

     // 4
     void InitializePatrolRoute()
     {
         // 5
         foreach(Transform child in patrolRoute)
         {
             // 6
             locations.Add(child);
         }
     }

     void OnTriggerEnter(Collider other)
     {
         // ... Изменения не требуются ...
     }

     void OnTriggerExit(Collider other)
     {
         // ... Изменения не требуются ...
     }
}
```

2. Выберите объект `Enemy` и перетащите объект `Patrol Route` из окна Hierarchy в переменную `Patrol Route` в сценарий `EnemyBehavior` (рис. 9.6).
3. Нажмите значок стрелки рядом с переменной `Locations` в окне Inspector и запустите игру. Вы увидите заполненный список (рис. 9.7).

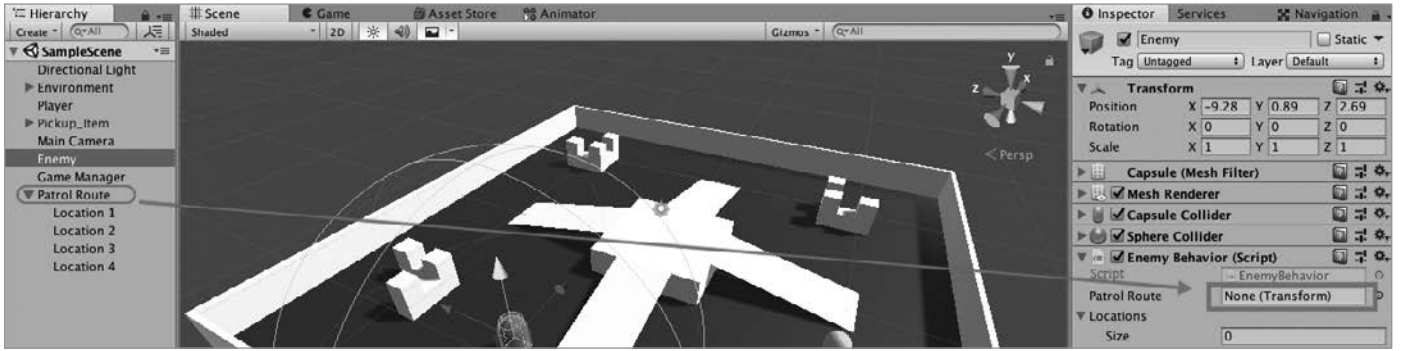

Hierarchy
Create
SampleScene
Directional Light
Environment
Player
Pickup_Item
Main Camera
Enemy
Game Manager
Patrol Route
Location 1
Location 2
Location 3
Location 4
Scene
Game
Asset Store
Animator
Shaded
2D
Gizmos
Persp
Inspector
Services
Navigation
Enemy
Static
Tag
Untagged
Layer
Default
Transform
Position
Rotation
Scale
Capsule (Mesh Filter)
Mesh Renderer
Capsule Collider
Sphere Collider
Enemy Behavior (Script)
Script
EnemyBehavior
Patrol Route
None (Transform)
Locations
Size

Рис. 9.6

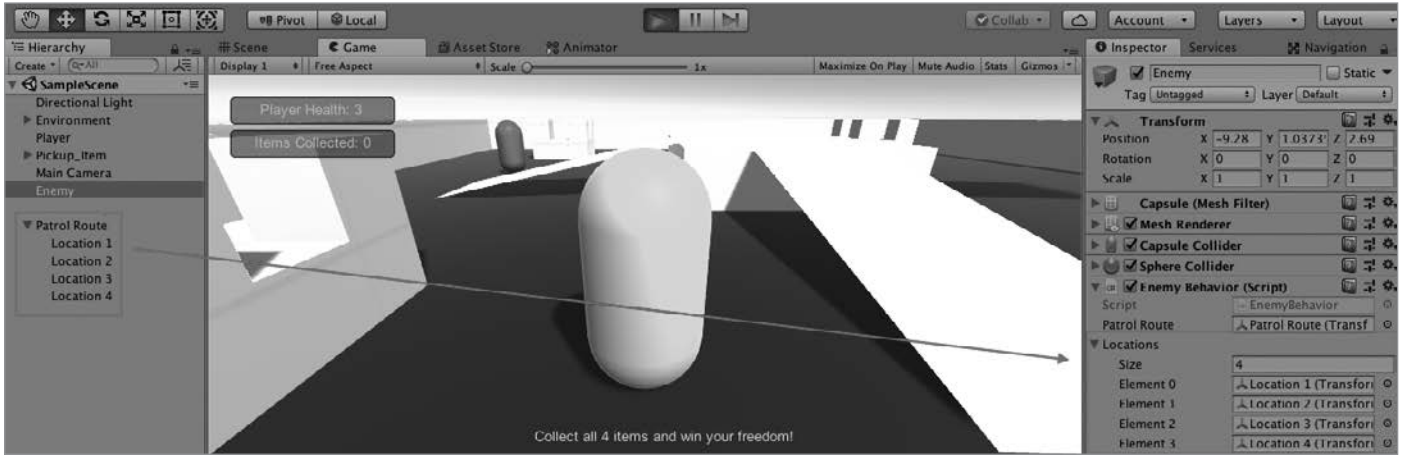

Pivot
Local
Collab
Account
Layers
Layout
Hierarchy
Create
SampleScene
Directional Light
Environment
Player
Pickup_Item
Main Camera
Enemy
Patrol Route
Location 1
Location 2
Location 3
Location 4
Scene
Game
Asset Store
Animator
Display 1
Free Aspect
Scale
Maximize On Play
Mute Audio
Stats
Gizmos
Player Health: 3
Items Collected: 0
Collect all 4 items and win your freedom!
Inspector
Services
Navigation
Enemy
Static
Tag
Untagged
Layer
Default
Transform
Position
Rotation
Scale
Capsule (Mesh Filter)
Mesh Renderer
Capsule Collider
Sphere Collider
Enemy Behavior (Script)
Script
EnemyBehavior
Patrol Route
Patrol Route (Transf
Locations
Size
4
Element 0
Location 1 (Transfor
Element 1
Location 2 (Transfor
Element 2
Location 3 (Transfor
Element 3
Location 4 (Transfor

Рис. 9.7

Разберем этот код.

1. Сначала объявим переменную для хранения пустого родительского объекта `patrolRoute`.
2. Затем объявляем переменную типа `List` для хранения всех дочерних компонентов `Transform` объекта `patrolRoute`.
3. После этого в методе `Start()` вызовем метод `InitializePatrolRoute()` в момент запуска игры.
4. Создаем `InitializePatrolRoute()` как приватный служебный метод для процедурного заполнения объекта `locarions` значениями `Transform`:
   - помните, что при отсутствии модификатора доступа переменные и методы становятся приватными по умолчанию.
5. Затем мы используем оператор `foreach` для перебора каждого дочернего `GameObject` в объекте `patrolRoute` и берем их компоненты `Transform`:
   - каждый компонент `Transform` записывается в локальной переменной `child`, объявленной в цикле `foreach`.
6. Наконец, перебирая дочерние объекты в `patrolRoute`, мы добавляем все `childTransform` в список `locations` с помощью метода `Add()`:
   - таким образом, независимо от того, какие изменения мы будем вносить на панели Hierarchy, список `locations` всегда будет заполнен всеми дочерними объектами из `patrolRoute`.

Мы могли бы и вручную занести каждый объект-локацию в список, перетаскивая их прямо с панели Hierarchy на панель Inspector, но такие соединения легко потерять или разорвать. Если мы позже внесем изменения в имена объектов, добавим или удалим объекты или поменяем еще что-нибудь, то все может пойти прахом. Гораздо безопаснее и удобнее процедурно заполнять списки или массивы `GameObject` в методе `Start()`.

По этой причине я также предпочитаю для поиска и сохранения ссылок на компоненты, прикрепленные к данному классу, использовать метод GetComponent() в методе Start(), а не назначать их на панели Inspector.

Теперь нам нужно, чтобы враг двигался по маршруту, который мы проложили. Это и будет вашей следующей задачей.

Время действовать. Перемещаем врага

Когда список точек маршрута известен и инициализирован в методе `Start()`, мы можем взять компонент `NavMeshAgent` врага и установить его в первую точку маршрута. Добавьте в сценарий `EnemyBehavior` следующий код и нажмите кнопку Play:

```
// 1
using UnityEngine.AI;

public class EnemyBehavior : MonoBehaviour
{
    public Transform patrolRoute;
    public List<Transform> locations;

    // 2
    private int locationIndex = 0;

    // 3
    private NavMeshAgent agent;

    void Start()
    {
        // 4
        agent = GetComponent<NavMeshAgent>();

        InitializePatrolRoute();

        // 5
        MoveToNextPatrolLocation();
    }

    void InitializePatrolRoute()
    {
        // ... Изменения не требуются ...
    }

    void MoveToNextPatrolLocation()
    {
        // 6
        agent.destination = locations[locationIndex].position;
    }

    void OnTriggerEnter(Collider other)
    {
```

```
        // ... Изменения не требуются ...
    }

    void OnTriggerExit(Collider other)
    {
        // ... Изменения не требуются ...
    }
}
```

Разберем этот код.

1. Мы подключили класс `UnityEngine.AI` с помощью директивы `using`, чтобы сценарий `EnemyBehaviour` имел доступ к классам навигации Unity. Если конкретно, то нам нужен `NavMeshAgent`.
2. Затем мы объявляем переменную для хранения точки, к которой в данный момент идет противник. Поскольку элементы в списке нумеруются с нуля, мы можем заставить префаб `Enemy` перемещаться между точками патрулирования в том порядке, в котором они хранятся в списке `locations`.
3. Объявляем переменную для хранения компонента `NavMeshAgent`, прикрепленного к объекту `Enemy`. Она будет приватной, поскольку никакие другие классы не должны иметь к нему доступ или изменять ее.
4. После этого с помощью метода `GetComponent()` мы находим и возвращаем прикрепленный компонент `NavMeshAgent` в переменную `agent`.
5. Затем вызываем метод `MoveToNextPatrolLocation()` в методе `Start()`.
6. Наконец, объявляем `MoveToNextPatrolLocation()` как приватный метод и устанавливаем значение `agent.destination`:
   - `destination` — это положение в трехмерном пространстве в формате `Vector3`;
   - вызов `locations[locationIndex]` захватывает элемент `Transform` по заданному индексу;
   - добавление `.position` позволяет сослаться на компонент `Transform` объекта.

Теперь при запуске сцены список `locations` заполняется точками патрулирования и вызывается метод `MoveToNextPatrolLocation()` для перемещения компонента `NavMeshAgent` на первый элемент в списке.

Следующим шагом будет поочередное перемещение вражеского объекта из первой точки во все остальные.

Время действовать. Воплощаем непрерывное патрулирование между локациями

Наш враг уже знает, где его первая точка патрулирования, но затем он останавливается. Мы же хотим, чтобы он постоянно перемещался по маршруту, что потребует дополнительной логики в методах `Update()` и `MoveToNextPatrolLocation()`. Создадим это поведение.

Добавьте в сценарий `EnemyBehavior` следующий код и нажмите кнопку Play:

```
public class EnemyBehavior : MonoBehaviour
{
    // ... Изменения не требуются ...

    void Start()
    {
        // ... Изменения не требуются ...
    }

    void Update()
    {
        // 1
        if(agent.remainingDistance < 0.2f && !agent.pathPending)
        {
            // 2
            MoveToNextPatrolLocation();
        }
    }

    void MoveToNextPatrolLocation()
    {
        // 3
        if (locations.Count == 0)
            return;

        agent.destination = locations[locationIndex].position;

        // 4
        locationIndex = (locationIndex + 1) % locations.Count;
    }

    // ... Другие изменения не требуются ...
}
```

Разберем этот код.

1. Мы объявляем метод `Update()` и добавляем оператор `if`, чтобы проверить, верны ли два условия:
   - свойство `remainingDistance` возвращает информацию о том, как далеко компонент `NavMeshAgent` находится от точки назначения;
   - свойство `pathPending` возвращает истину или ложь в зависимости от того, вычисляет ли Unity путь для компонента `NavMeshAgent`.
2. Если `agent` находится слишком близко к месту назначения и новый путь не вычисляется, то оператор `if` возвращает истину и вызывает метод `MoveToNextPatrolLocation()`.
3. Здесь мы добавили оператор `if`, чтобы убедиться, что список `locations` не пуст и можно спокойно выполнять метод `MoveToNextPatrolLocation()`:
   - если список `locations` пуст, то мы используем ключевое слово `return`, чтобы выйти из метода, не продолжая.

Такой метод называется защитным программированием, и в сочетании с рефакторингом он весьма полезен и должен быть в вашем арсенале, если вы переходите к сложной работе с языком C#.

4. Затем к значению `locationIndex` мы прибавляем `1` и применяем остаток от деления на `location.Count`:
   - индекс будет увеличиваться с 0 до 4, а затем снова станет равен 0, и траектория врага, таким образом, замкнется;
   - оператор модуля возвращает остаток от двух делимых значений — 2, деленное на 4, имеет остаток 2, поэтому 2 % 4 = 2. Аналогично 4, деленное на 4, не имеет остатка, поэтому 4 % 4 = 0.

Деление индекса на максимальное количество элементов в коллекции — классический способ найти следующий элемент. Но если вы забыли, как работает этот оператор, то вернитесь к главе 2.

Теперь нам нужно в каждом кадре метода `Update()` проверять, движется ли враг к своему установленному месту патрулирования. Когда он приближается, выполняем метод `MoveToNextPatrolLocation()`, который увеличивает `locationIndex` и задает следующую точку патрулирования в качестве пункта назначения. Если вы разместите панель Scene внизу, рядом с окном Console, как показано на рис. 9.8, и нажмете Play, то сможете наблюдать, как префаб `Enemy` перемещается по углам уровня.

Теперь враг движется по маршруту патрулирования внутри уровня, но не ищет и не атакует игрока, когда тот приближается к нему. Чтобы это исправить, используем компонент `NavAgent` в следующем разделе.

Механика врага

Теперь, когда наш `Enemy` непрерывно патрулирует периметр, пришло время дать ему собственную механику взаимодействия. Было бы скучно играть, если бы мы не дали врагу возможность напасть на нас.

Найти и уничтожить

В этом разделе мы реализуем переключение цели вражеского компонента `NavMeshAgent`, когда игрок приближается слишком близко, и нанесение урона в случае столкновения. Успешно «повреждая» игрока, враг будет возвращаться на свой маршрут патрулирования до следующей встречи с игроком. Однако мы не собираемся отдавать игрока на растерзание. Мы также добавим код для отслеживания здоровья врага, позволим понять, когда выстрел игрока попал по врагу и когда его необходимо уничтожить.

Время действовать. Меняем пункт назначения агента

Теперь, когда префаб `Enemy` перемещается по маршруту, нам нужно получить ссылку на позицию игрока и изменить пункт назначения `NavMeshAgent`.

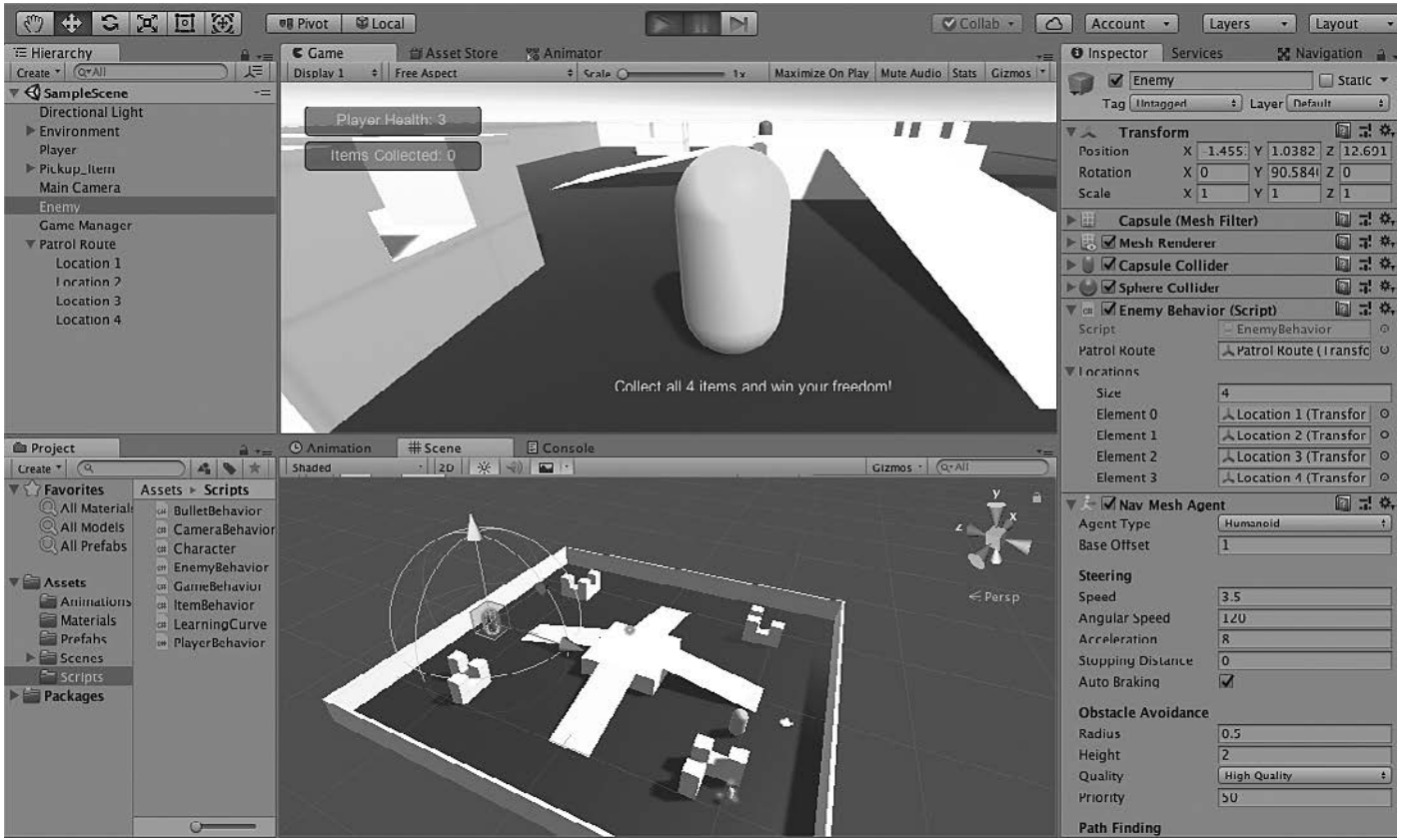

Pivot
Local
Collab
Account
Layers
Layout
Hierarchy
Create
SampleScene
Directional Light
Environment
Player
Pickup_Item
Main Camera
Enemy
Game Manager
Patrol Route
Location 1
Location 2
Location 3
Location 4
Game
Asset Store
Animator
Display 1
Free Aspect
Scale
Maximize On Play
Mute Audio
Stats
Gizmos
Player Health: 3
Items Collected: 0
Collect all 4 items and win your freedom!
Inspector
Services
Navigation
Enemy
Static
Tag
Untagged
Layer
Default
Transform
Position
Rotation
Scale
Capsule (Mesh Filter)
Mesh Renderer
Capsule Collider
Sphere Collider
Enemy Behavior (Script)
Script
EnemyBehavior
Patrol Route
Locations
Size
Element 0
Element 1
Element 2
Element 3
Nav Mesh Agent
Agent Type
Humanoid
Base Offset
Steering
Speed
Angular Speed
Acceleration
Stopping Distance
Auto Braking
Obstacle Avoidance
Radius
Height
Quality
High Quality
Priority
Path Finding
Project
Favorites
All Materials
All Models
All Prefabs
Assets
Animations
Materials
Prefabs
Scenes
Scripts
Packages
Assets ▸ Scripts
BulletBehavior
CameraBehavior
Character
EnemyBehavior
GameBehaviour
ItemBehavior
LearningCurve
PlayerBehavior
Animation
Scene
Console
Shaded
2D
Gizmos
Persp

Рис. 9.8

Добавьте в сценарий EnemyBehavior следующий код и нажмите кнопку Play:

```
public class EnemyBehavior : MonoBehaviour
{
    // 1
    public Transform player;

    public Transform patrolRoute;
    public List<Transform> locations;

    private int locationIndex = 0;
    private NavMeshAgent agent;

    void Start()
    {
        agent = GetComponent<NavMeshAgent>();

        // 2
        player = GameObject.Find("Player").transform;

        // ... Другие изменения не требуются ...
    }

    /* ... Нет изменений в Update,
           InitializePatrolRoute или
           MoveToNextPatrolLocation ... */

    void OnTriggerEnter(Collider other)
    {
        if(other.name == "Player")
        {
            // 3
            agent.destination = player.position;

            Debug.Log("Enemy detected!");
        }
    }

    void OnTriggerExit(Collider other)
    {
        // ... Изменения не требуются ...
    }
}
```

Разберем этот код.

1. Сначала мы объявили публичную переменную для хранения значения компонента Transform капсулы игрока.

2. Затем мы используем метод `GameObject.Find("Player")`, чтобы вернуть ссылку на объект игрока в сцене:
   - добавление приписки `.transform` позволяет напрямую сослаться на компонент `Transform` в той же строке.
3. Наконец, мы устанавливаем значение `agent.destination` на положение игрока в методе `OnTriggerEnter()` всякий раз, когда игрок входит в зону атаки врагов.

Если вы запустите игру и подойдете слишком близко к нашему недремлющему врагу, то он резко изменит намерения, сойдет со своего пути и направится к вам. Как только он настигнет игрока, заработает код в методе `Update()` и префаб `Enemy` вернется на маршрут.

Нам все еще нужно, чтобы враг мог каким-то образом навредить игроку, и ниже мы узнаем, как это сделать.

Время действовать. Снижаем здоровье игрока

Мы уже приложили немало усилий, чтобы создать врага, но пока нам все еще скучно, ведь ничего не происходит, когда префаб `Enemy` сталкивается с префабом `Player`. Чтобы это исправить, мы свяжем новую механику врага с менеджером игры.

Добавьте в сценарий `PlayerBehavior` следующий код и нажмите Play:

```
public class PlayerBehavior : MonoBehaviour
{
    // ... Изменение публичных переменных не требуется ...

    private float _vInput;
    private float _hInput;
    private Rigidbody _rb;
    private CapsuleCollider _col;

    // 1
    private GameBehavior _gameManager;

    void Start()
    {
        _rb = GetComponent<Rigidbody>();
        _col = GetComponent<CapsuleCollider>();
```

```
        // 2
        _gameManager = GameObject.Find("Game
            Manager").GetComponent<GameBehavior>();
    }

    /* ... Нет изменений в Update,
           FixedUpdate или
           IsGrounded ... */

    // 3
    void OnCollisionEnter(Collision collision)
    {
        // 4
        if(collision.gameObject.name == "Enemy")
        {
            // 5
            _gameManager.HP -= 1;
        }
    }
}
```

Разберем этот код.

1. Объявляем приватную переменную для хранения ссылки на экземпляр `GameBehavior` на сцене.
2. Затем находим и возвращаем сценарий `GameBehavior`, прикрепленный к объекту `Game Manager` на сцене:
   - использование метода `GetComponent()` в той же строке, что и `GameObject.Find()`, позволяет избавиться от лишнего кода.
3. Поскольку именно с объектом `Player` будет сталкиваться враг, имеет смысл объявить метод `OnCollisionEnter()` в сценарии `PlayerBehavior`.
4. Затем мы проверяем имя сталкивающегося объекта; если это префаб `Enemy`, то выполняем оператор `if`.
5. Наконец, вычитаем `1` из здоровья игрока с помощью экземпляра `_gameManager`.

Каждый раз, когда враг находит и нападает на игрока, менеджер игры активизирует свойство `set` параметра `HP`. В интерфейсе появится новое значение здоровья игрока; это значит, что и у нас появилась возможность добавить некую дополнительную логику условия поражения, но чуть позже.

Время действовать. Обнаруживаем столкновения с пулями

Теперь, когда у нас есть условие поражения, пришло время дать нашему игроку возможность отбиться от врага.

Измените код сценария `EnemyBehaviour`:

```
public class EnemyBehavior : MonoBehaviour
{
    public Transform player;
    public Transform patrolRoute;
    public List<Transform> locations;

    private int locationIndex = 0;
    private NavMeshAgent agent;

    // 1
    private int _lives = 3;
    public int EnemyLives
    {
        // 2
        get { return _lives; }

        // 3
        private set
        {
            _lives = value;

            // 4
            if (_lives <= 0)
            {
                Destroy(this.gameObject);
                Debug.Log("Enemy down.");
            }
        }
    }

    /* ... Нет изменений в Start,
           Update,
           InitializePatrolRoute,
           MoveToNextPatrolLocation,
           OnTriggerEnter или
           OnTriggerExit ... */

    void OnCollisionEnter(Collision collision)
    {
        // 5
        if(collision.gameObject.name == "Bullet(Clone)")
        {
```

```
            // 6
            EnemyLives -= 1;
            Debug.Log("Critical hit!");
        }
    }
}
```

Разберем этот код.

1. Мы объявили приватную переменную под названием `_lives`, публичный аналог которой называется `EnemyLives`. Это позволит нам управлять здоровьем врага, как мы уже делали в сценарии `GameBehavior`.
2. Затем мы говорим свойству `get` всегда возвращать `_lives`.
3. Далее мы используем `private set`, чтобы связать новое значение `EnemyLives` с `_lives`.

До этого мы не использовали варианты private set и get, но у них могут быть свои модификаторы доступа, как и у любого другого исполняемого кода. Такое объявление означает, что только родительский класс будет иметь доступ к свойствам.

4. Затем мы добавляем оператор `if`, в котором проверим, стало ли значение `_lives` или равно `0`, означающее, что `Enemy` должен быть мертв:
   - в этом случае мы уничтожаем `GameObject Enemy` и выводим сообщение в консоль.
5. Поскольку с пулями сталкивается именно `Enemy`, разумно проверять эти столкновения в сценарии `EnemyBehavior` в методе `OnCollisionEnter()`.
6. Наконец, если имя сталкивающегося объекта соответствует пуле, мы уменьшаем `EnemyLives` на `1` и выводим другое сообщение.

Обратите внимание, что мы проверяем имя "Bullet(Clone)", хотя префаб называется Bullet. Дело в том, что Unity добавляет суффикс (Clone) к любому объекту, созданному с помощью метода Instantiate(), который как раз и используется в нашей логике стрельбы.

Можно также проверять тег GameObject, но это уже метод, присущий именно Unity, а мы предпочитаем работать с чистым C#.

Теперь игрок может дать отпор, когда враг попытается забрать его жизнь. Выстрелив во врага трижды, мы уничтожим его. И снова использование нами свойств `get` и `set` для обработки дополнительной логики оказывается гибким и масштабируемым решением. Нам осталось сделать последний шаг — добавить игровому менеджеру условие поражения.

Время действовать. Обновляем игровой менеджер

Чтобы полностью реализовать условие поражения, нам нужно обновить класс менеджера.

Откройте сценарий `GameBehavior` и добавьте в него следующий код. Затем дайте префабу `Enemy` трижды столкнуться с вами:

```
public class GameBehavior : MonoBehaviour
{
    public string labelText = "Collect all 4 items and win your
    freedom!";
    public int maxItems = 4;
    public bool showWinScreen = false;

    // 1
    public bool showLossScreen = false;

    private int _itemsCollected = 0;
    public int Items
    {
        // ... Изменения не требуются ...
    }

    private int _playerHP = 3;
    public int HP
    {
        get { return _playerHP; }
        set {
            _playerHP = value;

            // 2
            if(_playerHP <= 0)
            {
                labelText = "You want another life with that?";
                showLossScreen = true;
                Time.timeScale = 0;
            }
            else
```

```
            {
                labelText = "Ouch... that's got hurt.";
            }
        }
    }

    void OnGUI()
    {
        // ... Изменения не требуются ...

        // 3
        if(showLossScreen)
        {
            if (GUI.Button(new Rect(Screen.width / 2 - 100,
              Screen.height / 2 - 50, 200, 100), "You lose..."))
             {
                SceneManager.LoadScene(0);
                Time.timeScale = 1.0f;
            }
        }
    }
}
```

Разберем этот код.

- Сначала мы объявляем переменную `public bool`, в которой хранится информация о том, когда в интерфейсе должен появиться экран поражения.
- Затем добавляем оператор `if`, чтобы поймать момент, когда значение `_playerLives` падает ниже `0`:
  - если это произошло, то меняем `labelText`, `showLossScreen`, а также `Time.timeScale`;
  - если игрок остался жив после столкновения с врагом, то `labelText` выводит другое сообщение.
- Наконец, мы постоянно проверяем, истинно ли условие `showLossScreen`, после чего создаем и выводим кнопку того же размера, что и кнопка победы, но с другим текстом:
  - когда пользователь нажимает кнопку поражения, уровень перезапускается, значение `timeScale` сбрасывается на `1` и управление возвращается к игроку.

Вот и все! Мы успешно добавили «умного» врага, который может нанести урон игроку и получить сдачи, а также добавили экран поражения. Прежде чем мы закончим эту главу, нам нужно обсудить еще одну

важную тему — как избежать написания повторяющегося кода. Такой код — проклятие для всех программистов, поэтому лучше как можно раньше научиться избавляться от него!

Рефакторинг — держим код в чистоте

Аббревиатура DRY (Don't Repeat Yourself) — главная мантра разработчика программного обеспечения. Она подсказывает, когда вы принимаете плохое или сомнительное решение, и позволяет чувствовать удовлетворение после хорошо выполненной работы.

На практике повторяющийся код используется в программировании довольно часто. Попытка избежать повторов и постоянные размышления о будущем создадут в вашем проекте столько сложностей, что даже не захочется его доделывать. Более эффективно и разумно подойти к работе с повторяющимся кодом так: опознать места, где он возникает, а затем найти способ удалить его. Данная задача называется рефакторингом. Применим капельку этого волшебства к классу `GameBehavior`.

Время действовать. Создаем метод перезапуска

Чтобы провести рефакторинг кода перезапуска уровня, измените сценарий `GameBehavior` следующим образом:

```
public class GameBehavior : MonoBehaviour
{
    // ... Изменения не требуются ...

    // 1
    void RestartLevel()
    {
        SceneManager.LoadScene(0);
        Time.timeScale = 1.0f;
    }

    void OnGUI()
    {
        GUI.Box(new Rect(20, 20, 150, 25), "Player Health: " +
          _playerLives);
        GUI.Box(new Rect(20, 50, 150, 25), "Items Collected: " +
          _itemsCollected);
        GUI.Label(new Rect(Screen.width / 2 - 100, Screen.height - 50,
          300, 50), labelText);
```

```
        if (showWinScreen)
        {
            if (GUI.Button(new Rect(Screen.width/2 - 100,
            Screen.height/2 - 50, 200, 100), "YOU WON!"))
        {
                // 2
                RestartLevel();
            }
        }

        if(showLossScreen)
        {
            if (GUI.Button(new Rect(Screen.width / 2 - 100,
            Screen.height / 2 - 50, 200, 100), "You lose..."))
            {
                RestartLevel();
            }
        }
    }
}
```

Разберем этот код.

1. Сначала объявим приватный метод `RestartLevel()`, который будет выполнять тот же код, что и кнопки выигрыша/проигрыша в методе `OnGUI()`.
2. Затем заменяем оба экземпляра кода перезапуска в `OnGUI()` вызовом метода `RestartLevel()`.

Проанализировав код, вы всегда найдете места, где требуется рефакторинг. Последней, хоть и необязательной вашей задачей будет рефакторинг логики победы/поражения в менеджере игры, что мы и сделаем ниже.

Испытание героя. Рефакторим логику выигрыша/проигрыша

Пока ваше сознание настроено на рефакторинг, вы могли заметить, что код, обновляющий значение `labelText`, `showWinScreen`/`showLossScreen` и `Time.timeScale`, повторяется в блоке `set` и у `Lives`, и у `Items`. Задача такова: написать приватную вспомогательную функцию, которая принимает параметры для каждой из этих переменных и присваивает им значения. Затем замените повторяющийся код в объектах `Lives` и `Items` с помощью вызова нового метода. Удачного кодирования!

Подведем итоги

На этом взаимодействие `Enemy` и `Player` можно считать готовым. Мы умеем наносить урон, а также получать его, терять жизни и бросать врагу вызов, и во всех этих случаях у нас обновляется интерфейс. С помощью навигационной системы Unity наши враги ходят по арене и переходят в режим атаки, находясь на определенном расстоянии от игрока. Каждый `GameObject` отвечает за свое поведение, внутреннюю логику и столкновения объектов, а игровой менеджер отслеживает переменные, которые управляют состоянием игры. Наконец, мы на простом примере поговорили о процедурном программировании и о том, как очистить код, если абстрагировать повторяющиеся инструкции в отдельных методах.

На данном этапе вы должны почувствовать удовлетворение, особенно если начали читать книгу, будучи абсолютным новичком. Освоить новый язык программирования в процессе создания полноценной рабочей игры — непростая задача. В следующей главе вы познакомитесь с некоторыми более сложными темами языка C#: с модификаторами типов, перегрузкой методов, интерфейсами и расширениями классов.

Контрольные вопросы. ИИ и навигация

1. Как в сцене Unity создается компонент `NavMesh`?
2. Какой компонент позволяет определить `GameObject` для `NavMesh`?
3. Примером какого метода программирования является реализация одной и той же логики на одном или нескольких объектах подряд?
4. Что означает аббревиатура DRY?

10 Снова о типах, методах и классах

Теперь, когда мы запрограммировали игровую механику и реализовали взаимодействие со встроенными классами Unity, пришло время углубить наши знания языка C# и поговорить о более сложных задачах, которые можно было бы решить, воспользовавшись нашими новыми знаниями. В этой главе мы вернемся к старым друзьям — переменным, типам, методам и классам, — но рассмотрим более сложные приложения и варианты использования. Многие из тем, которые мы рассмотрим, не будут касаться игры Hero Born в текущем ее состоянии, и поэтому отдельные примеры будут стоять обособленно, не касаясь непосредственно прототипа игры.

В данной главе вас ждет много новой информации, поэтому если в какой-то момент вы испытаете сложности, то смело возвращайтесь к первым нескольким главам, чтобы повторить тот или иной вопрос. Кроме того, здесь мы несколько оторвемся от игровых механик и функций, характерных для Unity, и обсудим такие темы, как:

- промежуточные модификаторы;
- перегрузка метода;
- использование параметров `out` и `ref`;
- работа с интерфейсами;
- абстрактные классы и переопределение;
- расширение функциональности класса;
- конфликты пространств имен;
- псевдонимы типов.

Подробнее о модификаторах доступа

Работая с переменными, мы уже привыкли добавлять модификаторы публичного и приватного доступа. Но существует также много других модификаторов, которых мы не видели. Мы не сможем в этой главе подробно рассмотреть их все, но те пять, на которых остановимся, позволят вам улучшить понимание языка C# и навыки программирования.

В этом разделе мы рассмотрим первые три модификатора из списка ниже, а оставшиеся два оставим для следующего раздела:

- `const`;
- `readonly`;
- `static`;
- `abstract`;
- `override`.

Полный список доступных модификаторов можно найти по ссылке docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/.

Свойства constant и readonly

Иногда в программе нужны переменные, в которых бы хранились постоянные и неизменяемые значения. Добавление к переменной ключевого слова `const` после модификатора доступа позволяет реализовать именно это, но только для встроенных типов C#. Например, значение переменной `maxItems` в классе `GameBehavior` стоило бы сделать постоянным:

```
public const int maxItems = 4;
```

Проблема работы с постоянными переменными заключается в том, что значение им можно присвоить только в момент объявления, то есть мы не можем задать `maxItems` без начального значения:

```
public readonly int maxItems;
```

Использование ключевого слова `readonly` для объявления переменной даст нам то же неизменяемое значение, что и константа, но при этом позволит присвоить ее начальное значение в любое время.

Использование ключевого слова static

Мы уже рассмотрели, как объекты или экземпляры создаются из схемы классов и что все свойства и методы после создания принадлежат этому конкретному экземпляру. Такой метод отлично подходит для объектно-ориентированной парадигмы, однако не все классы нужно создавать и не все свойства должны принадлежать конкретному экземпляру. При этом статические классы получаются «запечатанными»; это значит, что они не могут использоваться в наследовании классов.

Например, если нам надо определить несколько вспомогательных методов, нам нет нужды создавать экземпляр некоего класса `Utility`, поскольку в этом случае все методы класса логически не будут зависеть от конкретного объекта. Ваша задача — создать именно такой вспомогательный метод в новом сценарии.

Время действовать. Создаем статический класс

Создадим новый класс для хранения будущих (еще не написанных) методов, которые будут работать с сырыми вычислениями или повторяющейся логикой, не зависящей от игрового процесса.

1. Создайте новый сценарий C# в папке Scripts и назовите его `Utilities`.
2. Откройте его и добавьте следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```
// 1
using UnityEngine.SceneManagement;

// 2
public static class Utilities
{
    // 3
    public static int playerDeaths = 0;

    // 4
    public static void RestartLevel()
    {
        SceneManager.LoadScene(0);
        Time.timeScale = 1.0f;
    }
}
```

3. Удалите метод `RestartLevel()` из сценария `GameBehavior` и добавьте в метод `OnGUI()` следующий код:

```
void OnGUI()
{
    // ... Другие изменения не требуются ...

    if (showWinScreen)
    {
        if (GUI.Button(new Rect(Screen.width/2 - 100,
        Screen.height/2 - 50, 200, 100), "YOU WON!"))
        {
            // 5
            Utilities.RestartLevel();
        }
    }

    if(showLossScreen)
    {
        if (GUI.Button(new Rect(Screen.width / 2 - 100,
        Screen.height / 2 - 50, 200, 100), "You lose..."))
        {
            Utilities.RestartLevel();
        }
    }
}
```

Разберем этот код.

1. Сначала мы импортировали класс `SceneManagement`, чтобы получить доступ к методу `LoadScene()`.

2. Затем объявляем класс `Utilities` как публичный статический класс, который не наследуется от `MonoBehavior`, поскольку нам не нужно, чтобы он находился на игровой сцене.
3. Далее создаем публичную статическую переменную, в которой будет храниться информация о количестве смертей и перезапусков игры игроком.
4. Затем объявляем публичный статический метод для хранения логики перезапуска уровня, которая в настоящее время находится в сценарии `GameBehavior`.
5. Наконец, вызываем метод `RestartLevel()` класса `Utilities` при нажатии кнопки экрана победы или поражения. Обратите внимание: для вызова метода не нужен экземпляр класса `Utilities`, поскольку он статический. Достаточно просто точечной нотации.

Теперь мы отделили логику перезапуска от сценария `GameBehavior` и поместили ее в статический класс, что упрощает возможное повторное использование этого кода в проекте. Применение модификатора `static` также гарантирует, что нам не придется создавать экземпляры класса `Utilities` или управлять ими, прежде чем мы задействуем его члены класса.

Нестатические классы могут иметь статические и нестатические свойства и методы. Однако если весь класс является статическим, то все его свойства и методы тоже должны быть такими.

На этом мы закончим углубленное знакомство с переменными и типами, а это значит, что пора рассмотреть тему методов и их более сложных возможностей, включая перегрузку методов и параметры `ref` и `out`.

Вернемся к методам

С момента, когда в главе 3 вы узнали, что такое методы, они стали занимать большую часть кода, но существует еще два более интересных варианта использования, которые мы еще не рассмотрели: перегрузка метода и использование именованных параметров `ref` и `out`.

Перегрузка методов

Термин «перегрузка метода» означает создание нескольких методов с одним и тем же именем, но с разными сигнатурами. Сигнатура метода состоит из его имени и набора параметров, и именно по этим признакам компилятор C# отличает один метод от другого. В качестве примера возьмем следующий метод:

```
public bool AttackEnemy(int damage) {}
```

Сигнатура метода `AttackEnemy` выглядит так:

```
AttackEnemy(int)
```

Теперь, когда мы знаем сигнатуру метода `AttackEnemy`, его можно перегрузить, изменив количество параметров или типы параметров, оставив при этом то же имя. Это обеспечивает дополнительную гибкость, когда для одного и того же действия возможны разные варианты.

Метод `RestartLevel()` в сценарии `GameBehavior` отлично иллюстрирует ситуацию, когда мы могли бы использовать перегрузку метода. В данный момент метод `RestartLevel()` перезапускает только текущий уровень, но что, если у нас будет игра на сотню уровней или сцен? Можно было бы провести рефакторинг кода `RestartLevel()` и научить его принимать параметры, но это часто приводит к излишнему разрастанию кода и путанице.

И опять же именно на методе `RestartLevel()` вы опробуете ваши новые знания. Ваша задача — перегрузить метод, чтобы он мог принимать разные параметры.

Время действовать. Перезагружаем метод перезапуска уровня

Напишем перегруженную версию метода `RestartLevel()`.

1. Откройте класс `Utilities` и добавьте в него следующий код:

```
public static class Utilities
{
    public static int playerDeaths = 0;
```

```
    public static void RestartLevel()
    {
        SceneManager.LoadScene(0);
        Time.timeScale = 1.0f;
    }

    // 1
    public static bool RestartLevel(int sceneIndex)
    {
        // 2
        SceneManager.LoadScene(sceneIndex);
        Time.timeScale = 1.0f;

        // 3
        return true;
    }
}
```

2. Откройте сценарий `GameBehavior` и обновите один из вызовов метода `Utilities.RestartLevel()` в методе `OnGUI()`:

```
if (showWinScreen)
 {
   if (GUI.Button(new Rect(Screen.width/2 - 100,
      Screen.height/2 - 50, 200, 100), "YOU WON!"))
   {
     // 4
     Utilities.RestartLevel(0);
   }
}
```

Разберем этот код.

1. Сначала мы объявляем перегруженную версию метода `RestartLevel()`, который теперь принимает параметр `int` и возвращает логическое значение.
2. Затем вызываем метод `LoadScene()` и передаем параметр `sceneIndex` вместо жесткого кодирования этого значения.
3. Метод возвращает `true` после загрузки новой сцены, и свойство `timescale` сбрасывается.
4. Наконец, вызываем перегруженный метод `RestartLevel()` и переходим на сцену с `sceneIndex = 0` при нажатии кнопки победы. Перегруженные методы автоматически обнаруживаются Visual Studio и отображаются по номерам, как показано на рис. 10.1.

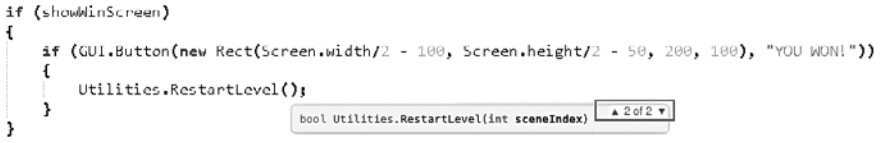


Рис. 10.1

Функциональные возможности метода `RestartLevel()` теперь стали более гибкими, и метод позволяет учитывать дополнительные ситуации, которые могут вам понадобиться позже.

Перегрузка метода не ограничивается статическими методами, которые были рассмотрены в предыдущем примере. Любой метод можно перегрузить, если его сигнатура отличается от оригинала.

Параметр ref

Когда еще в главе 5 мы говорили о классах и структурах, выяснилось, что не все объекты передаются одинаково. Объекты типа «значение» передаются по копии, а ссылочные типы передаются по ссылке. Но мы не говорили о том, как используются объекты или значения при передаче их в методы в качестве аргументов.

По умолчанию все аргументы передаются по значению, то есть переменная, переданная в метод, не изменится, даже если мы поменяем ее значение внутри тела метода. В большинстве случаев нас это устраивает, но бывают ситуации, когда требуется передать аргумент метода по ссылке. При объявлении параметра нужно добавить перед его именем ключевое слово `ref` или `out`. В таком случае аргумент будет помечен как ссылка.

Важно упомянуть несколько ключевых моментов, о которых следует помнить при использовании ключевого слова `ref`:

- перед передачей в метод у аргумента должно быть задано значение;
- не требуется инициализировать или назначать значение ссылочного параметра перед завершением метода;

- аргументами `ref` или `out` не могут быть свойства для записи или чтения.

Попробуем добавить в игру отслеживание того, сколько раз игрок перезапускал игру.

Время действовать. Отслеживаем перезапуски со стороны игрока

Создадим метод, который будет обновлять значение переменной `playerDeaths`, чтобы посмотреть, как работают аргументы метода, передающиеся по ссылке.

Добавьте в класс `Utilities` следующий код:

```
public static class Utilities
{
    public static int playerDeaths = 0;

    // 1
    public static string UpdateDeathCount(ref int countReference)
    {
        // 2
        countReference += 1;
        return "Next time you'll be at number " + countReference;
    }

    public static void RestartLevel()
    {
        SceneManager.LoadScene(0);
        Time.timeScale = 1.0f;

        // 3
        Debug.Log("Player deaths: " + playerDeaths);
        string message = UpdateDeathCount(ref playerDeaths);
        Debug.Log("Player deaths: " + playerDeaths);
    }

    public static bool RestartLevel(int sceneIndex)
    {
        // ... Изменения не требуются ...
    }
}
```

Разберем этот код.

1. Сначала мы объявляем новый статический метод, который возвращает тип `string`, а принимает `int`, переданный по ссылке.
2. Затем напрямую обновляем ссылочный параметр, увеличивая его значение на 1, и возвращаем строку, содержащую новое значение.
3. Наконец, выводим переменную `playerDeaths` в методе `RestartLevel()` до и после передачи по ссылке методу `UpdateDeathCount()`.

Теперь если вы запустите игру и проиграете, то в консоли будет видно, что переменная `playerDeaths` увеличилась на 1 внутри метода `UpdateDeathCount()`, поскольку она была передана по ссылке, а не по значению (рис. 10.2):

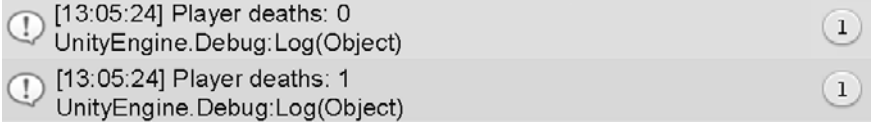


Рис. 10.2

В данном примере мы используем ключевое слово ref, но можно было обновить значение playerDeaths прямо внутри UpdateDeathCount() или добавить увеличение значения в методе RestartLevel() в случаях, когда перезапуск произошел именно по причине поражения.

Теперь, когда мы знаем, как использовать параметр `ref` в нашем проекте, рассмотрим параметр `out` и его предназначение.

Параметр out

Ключевое слово `out` выполняет ту же работу, что и `ref`, но работает несколько иначе:

- аргументы перед передачей в метод не нужно инициализировать;
- значение параметра, на которое указывает ссылка, необходимо инициализировать или присвоить в вызываемом методе перед возвратом.

Например, в методе `UpdateDeathCount()` мы могли бы использовать параметр `out` вместо `ref`, если значение параметра `countReference` будет задано внутри метода:

```
public static string UpdateDeathCount(out int countReference)
{
    countReference = 1;
    return "Next time you'll be at number " + countReference;
}
```

Методы, в которых используется ключевое слово `out`, лучше подходят для ситуаций, когда вам нужно вернуть несколько значений из одной функции, а ключевое слово `ref` работает лучше в случаях, когда нужно просто изменить ссылочное значение.

Вооружившись новыми возможностями методов, теперь мы можем вернуться к самой серьезной теме: объектно-ориентированному программированию (ООП). Эта тема столь обширна, что невозможно рассказать обо всем в одной или двух главах, но я опишу несколько ключевых инструментов, которые будут вам полезны на ранних этапах вашей карьеры разработчика. ООП — одна из тем, над которой вы можете продолжать работать после прочтения данной книги.

Подробнее об ООП

Объектно-ориентированное мышление необходимо для создания значимых приложений и понимания принципов «внутренней» работы языка C#. Сложность в том, что классы и структуры в ООП и проектировании объектов не являются самоцелью. Они лишь строительные блоки вашего кода. Проблема в том, что классы ограничены одиночным наследованием, то есть они могут иметь только один родительский класс, а структуры не могут наследовать вообще. Тогда вопрос, который перед нами встает, можно сформулировать просто: *«Как можно создавать объекты из одного и того же шаблона, но заставить их выполнять разные действия в зависимости от конкретного сценария?»*

Интерфейсы

Один из способов объединять объекты по их функционалу — это интерфейсы. Как и классы, интерфейсы представляют собой схемы передачи данных и описание поведения, но с одним важным отличием: у них нет реальной логики реализации и хранения значений. Класс или структура, реализующая интерфейс, должна заполнить значения и методы, указанные в интерфейсе. Самое замечательное в интерфейсах то, что их могут использовать как классы, так и структуры, и любой объект может реализовывать сколько угодно интерфейсов.

Предположим, мы хотим, чтобы враг мог стрелять в нашего игрока, если тот находится достаточно близко. Мы могли бы создать родительский класс, от которого наследовал бы и игрок, и враг, тем самым реализовав их создание по одному образцу. Однако проблема такого подхода заключается в том, что поведение и данные у врага и игрока не всегда одинаковые.

Более эффективный способ решить подобную задачу — определить интерфейс, в котором было бы изложено, что должен делать стреляющий объект, а затем реализовать этот интерфейс и во врага, и в игрока. В результате они по-прежнему существуют в разных классах, но получают общий функционал.

Время действовать. Создаем интерфейс менеджера

Рефакторинг механики стрельбы в интерфейс я оставлю на ваше самостоятельное выполнение, но нам все равно нужно рассмотреть, как создавать и реализовывать интерфейсы в коде. В этом примере мы создадим гипотетический интерфейс, который необходимо реализовать для всех сценариев менеджера, чтобы создать общую структуру.

Создайте новый сценарий C# в папке Scripts, назовите его `IManager` и добавьте в него следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```
// 1
public interface IManager
{
    // 2
    string State { get; set; }

    // 3

    void Initialize();
}
```

Разберем этот код.

1. Сначала мы объявляем открытый интерфейс под названием `IManager` с помощью ключевого слова `interface`.
2. Затем добавляем в `IManager` строковую переменную под названием `State` с модификаторами `get` и `set` для хранения текущего состояния принимающего класса.

Для всех свойств интерфейса для компиляции требуется как минимум модификатор get, но при необходимости можно задавать и get, и set.

3. Наконец, определяем метод с именем `Initialize()` без возвращаемого типа для реализации принимающим классом.

Ваша следующая задача — использовать интерфейс `IManager`, а это значит, что он должен быть реализован другим классом.

Время действовать. Используем интерфейс

Чтобы не усложнять задачу, попросим игровой менеджер реализовать наш новый интерфейс и применить его схему.

Добавьте в сценарий `GameBehavior` следующий код:

```
// 1
public class GameBehavior : MonoBehaviour, IManager
{
    // 2
    private string _state;
```

```
    // 3
    public string State
    {
        get { return _state; }
        set { _state = value; }
    }

    // ... Другие изменения не требуются ...

    // 4
    void Start()
    {
        Initialize();
    }

    // 5
    public void Initialize()
    {
        _state = "Manager initialized..";
        Debug.Log(_state);
    }

    void OnGUI()
    {
        // ... Изменения не требуются ...
    }
}
```

Разберем этот код.

1. Сначала мы объявляем, что сценарий `GameBehavior` реализует интерфейс `IManager`, используя запятую и его имя, как и при создании подклассов.
2. Затем добавляем приватную переменную, которую будем применять для хранения публичного значения `State`, реализации которой требует `IManager`.
3. Затем добавляем публичную переменную `State` из `IManager` и в качестве ее приватной альтернативы задаем `_state`.
4. После этого объявляем метод `Start()` и вызываем метод `Initialize()`.
5. Наконец, объявляем метод `Initialize()`, объявленный в `IManager`, с реализацией, которая задает и выводит публичную переменную `State`.

При этом мы указали, что `GameBehavior` реализует интерфейс `IManager`, то есть реализует его члены `State` и `Initialize()`, как показано на рис. 10.3.

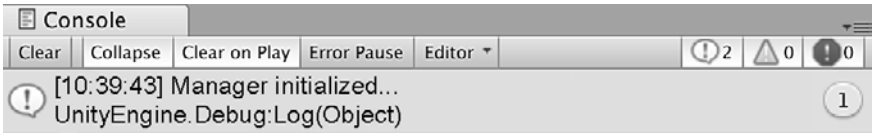


Рис. 10.3

Во многом этот код основан на том, что его реализация специфична для сценария `GameBehavior`. Если бы у нас был другой класс менеджера, то мы могли бы реализовать то же самое, но с другой логикой. Таким образом, перед нами открывается море возможностей для создания классов, одной из которых являются абстрактные классы.

Абстрактные классы

Абстрактный класс — еще один способ разделения общих схем создания и их совместного использования разными объектами. Как и интерфейсы, абстрактные классы не содержат логику реализации своих методов, но могут хранить значения переменных. Любой класс, который наследует от абстрактного класса, должен полностью реализовывать все переменные и методы, отмеченные ключевым словом `abstract`. Абстрактные классы бывают особенно полезны в случаях, когда вы хотите использовать наследование классов, не создавая реализацию метода по умолчанию для базового класса.

Например, возьмем функциональность интерфейса `IManager`, которую мы только что написали, и превратим его в абстрактный базовый класс:

```
// 1
public abstract class BaseManager
{
    // 2
    protected string _state;
    public abstract string state { get; set; }

    // 3
    public abstract void Initialize();
}
```

Разберем этот код.

1. Сначала мы объявляем новый класс под названием `BaseManager` с ключевым словом `abstract`.
2. Затем создаем две переменные:
   - строку с модификатором `protected` и именем `_state`, к которой могут получать доступ только классы, наследующиеся от `BaseManager`;
   - `abstract string` с именем `state` и методами `get` и `set`, которые будут реализованы подклассом.
3. Наконец, добавляем абстрактный метод `Initialize()`, который также должен быть реализован в подклассе.

Теперь схема класса `BaseManager` такая же, что и у `IManager`, и она позволяет любым подклассам определять свои реализации `state` и `Initialize()` с помощью ключевого слова `override`:

```
// 1
public class CombatManager: BaseManager
{
    // 2
    public override string state
    {
        get { return _state; }
        set { _state = value; }
    }

    // 3
    public override void Initialize()
    {
        _state = "Manager initialized..";
        Debug.Log(_state);
    }
}
```

Этот код работает следующим образом.

1. Сначала мы объявляем новый класс с именем `CombatManager`, который наследуется от абстрактного класса `BaseManager`.
2. Затем добавляем реализацию переменной `state` из `BaseManager` с помощью ключевого слова `override`.

3. Наконец, добавляем реализацию метода `Initialize()` из класса `BaseManager` с помощью ключевого слова `override` еще раз и задаем защищенную переменную `_state`.

Это лишь верхушка айсберга возможностей интерфейсов и абстрактных классов, но знать о них вы должны обязательно. Интерфейсы позволяют вам распределять и совместно использовать ту или иную функциональность в разных не связанных друг с другом объектах, и код в итоге собирается подобно конструктору лего.

С другой стороны, абстрактные классы дают возможность придерживаться структуры ООП с единичным наследованием и отделять реализацию класса от его схемы. Эти подходы можно даже смешивать и согласовывать, поскольку абстрактные классы могут реализовывать интерфейсы точно так же, как неабстрактные.

Как и во всех сложных темах, никогда не лишним будет обратиться к документации. Загляните на docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/abstract и docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/interface.

Не всегда есть необходимость создавать новый класс с нуля. Иногда достаточно добавить нужную функцию или логику к существующему классу — это называется расширением класса.

Расширения класса

Отвлечемся от пользовательских объектов и поговорим о том, как можно расширить существующие классы в случае роста наших потребностей. Идея проста: мы берем существующий встроенный класс C# и добавляем в него любые функции, необходимые нам. Поскольку у нас нет доступа к базовому коду, на котором написан язык C#, это единственный способ добавить что-то новое к объектам, уже имеющимся в языке.

Классы можно изменять только путем добавления методов. Использование переменных или других сущностей не допускается. Это несколько связывает нам руки, но зато делает синтаксис согласованным:

```
public static returnType MethodName(this ExtendingClass localVal) {}
```

Методы расширения объявляются с помощью того же синтаксиса, что и обычные методы, но с некоторыми оговорками.

- Все методы расширения должны быть помечены модификатором `static`.
- Первым параметром должно быть ключевое слово `this`; за ним следует имя класса, который мы хотим расширить, и имя локальной переменной:
  - этот параметр позволяет компилятору идентифицировать метод как расширение и дает нам локальную ссылку на экземпляр существующего класса;
  - после этого доступ к методам и свойствам класса можно будет получать через локальную переменную.
- Обычно методы расширения хранятся внутри статического класса, который, в свою очередь, хранится внутри своего пространства имен. Это позволяет вам контролировать, какие сценарии получают доступ к вашим функциям.

Ваша следующая задача — применить расширения класса на практике, добавив новый метод во встроенный C#-класс `String`.

Время действовать. Расширяем класс String

Опробуем расширение на практике, добавив собственный метод в класс `String`.

Создайте новый сценарий C# в папке Scripts, назовите его `CustomExtensions` и добавьте в него следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// 1
namespace CustomExtensions
{
    // 2
    public static class StringExtensions
    {
```

```
        // 3
        public static void FancyDebug(this string str)
        {
            // 4
            Debug.LogFormat("This string contains {0} characters.",
            str.Length);
        }
    }
}
```

Разберем этот код.

1. Сначала мы объявляем пространство имен `CustomExtensions`, в котором будут храниться все классы и методы расширения.
2. Затем объявляем статический класс по имени `StringExtensions` для «внутренних» целей. Эти действия необходимы для любой группы расширений класса.
3. Затем добавляем статический метод `FancyDebug` к классу `StringExtensions`:
   - первый параметр, `this string str`, отмечает метод как расширение;
   - параметр `str` будет содержать ссылку на фактическое текстовое значение, для которого вызывается метод `FancyDebug()`. Мы можем работать с типом `str` внутри тела метода как если бы он представлял собой любой строковый литерал.
4. Наконец, мы выводим сообщение отладки всякий раз, когда выполняется метод `FancyDebug`, используя метод `str.Length`, ссылаясь на строковую переменную, на которой вызывается метод.

Теперь, когда расширение для класса `String` готово, проверим его.

Время действовать. Пробуем метод расширения

Чтобы использовать наш новый строковый метод, нам нужно включить его в любой класс, которому нужен доступ к этому методу.

Откройте сценарий `GameBehavior` и напишите новый код класса:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// 1
using CustomExtensions;
```

```
public class GameBehavior : MonoBehaviour, IManager
{
    // ... Изменения не требуются ...

    void Start()
    {
        // ... Изменения не требуются ...
    }

    public void Initialize()
    {
        _state = "Manager initialized..";

        // 2
        _state.FancyDebug();

        Debug.Log(_state);
    }

    void OnGUI()
    {
        // ... Изменения не требуются ...
    }
}
```

Разберем этот код.

1. Сначала мы добавляем пространство имен `CustomExtensions` с помощью директивы `using` в верхней части файла.
2. Затем вызываем метод `FancyDebug` на строковой переменной `_state` с помощью точечной нотации внутри метода `Initialize()`, чтобы распечатать количество символов в строке.

Расширение всего строкового класса методом `FancyDebug()` означает, что любая строковая переменная получает к нему доступ. Поскольку первый параметр метода расширения ссылается на конкретное строковое значение, для которого вызывается метод `FancyDebug()`, его длина выводится правильно (рис. 10.4).

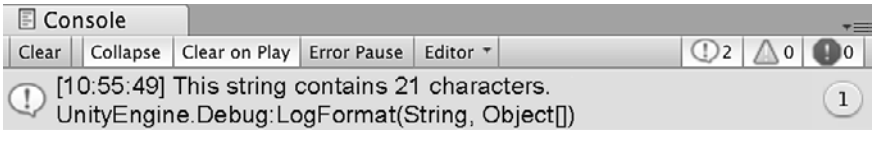


Рис. 10.4

Пользовательские классы можно расширять аналогичным образом, но чаще всего в этом случае мы просто добавляем нужные функции непосредственно в класс (при его наличии).

Последняя тема, которую мы рассмотрим в этой главе, — пространства имен, о которых ранее в книге мы говорили вскользь. В следующем разделе вы узнаете, насколько велика роль пространств имен в C# и о том, как создать псевдонимы типов.

И снова о пространствах имен

По мере того как ваши приложения будут усложняться, вы начнете разделять свой код на пространства имен, чтобы четко понимать и контролировать, где и когда к нему обращаются другие части кода. Вы также будете использовать сторонние программные инструменты и плагины, чтобы сэкономить время на написании с нуля функции, которую кто-то уже сделал до вас. Оба случая являются признаком того, что вы растете как программист, но они же могут стать причиной конфликта пространств имен.

Конфликты пространств имен возникают, когда у вас появляются несколько классов или типов с одним и тем же именем, и случается это чаще, чем нам бы хотелось. Если несколько программистов следуют общим правилам именования, то, как правило, это дает схожие результаты и у вас легко может появиться несколько классов с именами `Error` или `Extension`, а Visual Studio будет ругаться на это. К счастью, в C# есть простое решение для таких ситуаций: псевдонимы типов.

Псевдонимы типов

Определение псевдонима типа позволяет вам явно выбрать, какой из конфликтующих типов вы хотите использовать в данном классе, или создать более удобное имя вместо слишком длинного имеющегося

имени. Псевдонимы типов добавляются в верхней части файла класса с помощью директивы `using`, за которой следует имя псевдонима и назначенный тип:

```
using aliasName = type;
```

Например, если мы хотим создать псевдоним типа для имени типа `Int64`, то могли бы написать вот так:

```
using CustomInt = System.Int64;
```

Теперь имя `CustomInt` является псевдонимом для типа `System.Int64`, и компилятор будет работать с этим именем как с `Int64`, позволяя нам использовать его так же, как и любой другой тип:

```
public CustomInt playerHealth = 100;
```

Вы можете подменять псевдонимами и собственные типы, и уже существующие, при этом синтаксис будет одинаковым — объявление в верхней части файла сценария вместе с другими директивами `using`.

Подведем итоги

Теперь, владея новыми модификаторами, перегрузкой методов, расширениями классов и объектно-ориентированными навыками, вы находитесь всего в одном шаге от завершения нашего путешествия в C#. Помните: мы изучали эти более сложные темы для того, чтобы вы могли подумать об использовании знаний, которые получили в этой книге, в более сложных приложениях. Не стоит думать, будто материалы, приведенные в данной главе, охватывают эти темы целиком и полностью. Используйте новые знания как отправную точку и двигайтесь дальше.

В следующей главе мы обсудим общие понятия программирования, получим небольшой практический опыт работы с делегатами и событиями, а в конце обсудим обработку исключений.

Контрольные вопросы. Новый уровень!

1. Какое ключевое слово позволяет пометить переменную как неизменяемую?
2. Как создать перегруженную версию базового метода?
3. В чем заключается основная разница между классами и интерфейсами?
4. Как можно разрешить конфликт пространств имен в одном из ваших классов?

11 Знакомство со стеками, очередями и HashSet

В предыдущей главе мы еще раз обсудили переменные, типы и классы и рассмотрели их более сложный функционал по сравнению с тем, что изучали в начале книги. В этой главе мы в подробностях рассмотрим новые типы коллекций, а также их более сложные возможности.

У каждого из типов коллекций, которые будут представлены в этой главе, есть определенная цель. В большинстве сценариев, в которых вам нужно обработать набор данных, отлично подойдут списки и массивы. Если же нужно временное хранилище или возможность управлять порядком элементов коллекции, или, точнее, порядком доступа к ним, то подойдут стеки и очереди. Если вам нужно выполнять над коллекцией операции с учетом того, что каждый элемент в коллекции уникален, поможет `HashSet`.

Помните, что хороший программист не старается запомнить весь свой код, а выбирает правильный инструмент для правильной работы.

Прежде чем приступить к работе с кодом в следующем разделе, отметим темы, которые нам предстоит рассмотреть:

- введение в стеки;
- просмотр и извлечение элементов;
- общие методы;
- работа с очередями;
- добавление, удаление и просмотр элементов с помощью `HashSet`;
- выполнение операций.

Введение в стеки

Если говорить по-простому, то стек представляет собой множество элементов одного и того же типа. Длина у стека переменная, то есть меняется в зависимости от того, сколько элементов в нем содержится. Важное различие между стеком и списком или массивом заключается в том, как именно в нем хранятся элементы. Стеки следуют модели **last in first out (LIFO)**: последний введенный элемент в стеке является первым доступным для вывода. Это полезно, если нам нужно получить доступ к элементам в обратном порядке. Обратите внимание: в стеке могут храниться `null` и повторяющиеся значения.

Все типы коллекций, о которых мы говорим в данной главе, являются частью пространства имен System.Collections.Generic, поэтому не забудьте добавить следующий код в начало любого файла, в котором вы хотите их использовать:

```
using System.Collections.Generic;
```

Теперь, когда саму концепцию мы разобрали, взглянем на базовый синтаксис объявления стеков.

Базовый синтаксис

Объявление переменной стека выполняется следующим образом:

- ключевое слово `Stack`, тип элементов стека в треугольных скобках, а также уникальное имя объекта;
- ключевое слово `new` для инициализации стека в памяти, за которым следует ключевое слово `Stack` и тип элементов стека в треугольных скобках;
- пара круглых скобок `c` точкой с запятой в конце.

В общем виде это выглядит так:

```
Stack<elementType> name = new Stack<elementType>();
```

В отличие от других типов коллекций, с которыми вы работали, в стек нельзя поместить значения сразу в момент создания.

C# поддерживает необобщенную версию типа Stack, для которой не требуется определения типа элементов:

```
Stack myStack = new Stack();
```

Использовать такой тип менее безопасно и дороже, чем описанную выше версию. Почитайте рекомендации Microsoft на этот счет: github.com/dotnet/platform-compat/blob/master/docs/DE0006.md.

Ваша следующая задача — создать собственный стек и получить практический опыт работы с методами этого класса.

Время действовать. Храним собранные предметы

Чтобы проверить работу стеков, мы перепишем имеющуюся логику сбора предметов в игре Hero Born, используя стек для хранения собираемой добычи.

1. Откройте сценарий `GameBehavior.cs` и добавьте новую переменную стека с именем `lootStack`:

```
// 1
public Stack<string> lootStack = new Stack<string>();
```

2. Добавьте в метод `Initialize` следующий код для добавления новых элементов в стек:

```
public void Initialize()
{
    _state = "Manager initialized..";
    _state.FanceyDebug();
    Debug.Log(_state);

    // 2
    lootStack.Push("Sword of Doom");
    lootStack.Push("HP+");
    lootStack.Push("Golden Key");
    lootStack.Push("Winged Boot");
    lootStack.Push("Mythril Bracers");
}
```

3. Добавьте новый метод в конец сценария для вывода информации о стеке:

```
// 3
public void PrintLootReport()
```

```
{
    Debug.LogFormat("There are {0} random loot items waiting
    for you!", lootStack.Count);
}
```

4. Откройте `ItemBehavior.cs` и вызовите метод `PrintLootReport` из `gameManager`:

```
void OnCollisionEnter(Collision collision)
{
    if(collision.gameObject.name == "Player")
    {
        Destroy(this.transform.parent.gameObject);
        Debug.Log("Item collected!");
    }

    gameManager.Items += 1;

    // 4
    gameManager.PrintLootReport();
}
```

Разберем этот код.

1. Создаем пустой стек с элементами строкового типа.
2. Используем метод `Push` для добавления строковых элементов в стек, при этом его размер будет всякий раз увеличиваться.
3. Выводим количество элементов стека всякий раз, когда вызывается метод.
4. Вызываем метод `PrintLootReport` каждый раз, когда игрок подбирает предмет (рис. 11.1).

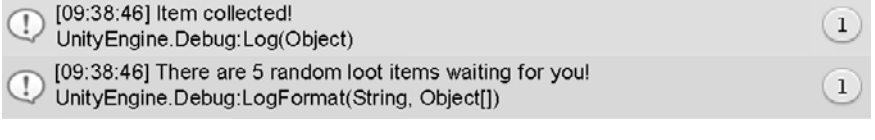


Рис. 11.1

Теперь, когда у нас есть рабочий стек, мы можем поэкспериментировать с тем, как можно доставать из стека элементы с помощью методов `Pop` и `Peek`.

Просмотр и извлечение

Мы уже говорили о том, что элементы в стеке хранятся по принципу LIFO. Теперь нам нужно узнать, как в этой коллекции осуществляется доступ к элементам — путем просмотра и извлечения:

- метод `Peek` возвращает следующий элемент в стеке, не удаляя его, то есть позволяет «посмотреть» на него, ничего не меняя;
- метод `Pop` возвращает и удаляет следующий элемент в стеке, по сути «выталкивая» его и передавая вам.

Оба метода можно использовать по отдельности или вместе, в зависимости от ваших целей. Далее мы получим практический опыт работы с обоими методами.

Время действовать. Получаем последний подобранный предмет

Ваша задача — получить последний элемент, добавленный в стек `lootStack`. В нашем примере такой элемент определяется программно в методе `Initialize`, но можно предположить, что элементы добавляются случайным образом или по какому-то иному принципу. В любом случае добавим в `PrintLootReport` следующий код:

```
public void PrintLootReport()
{
    // 1
    var currentItem = lootStack.Pop()

    // 2
    var nextItem = lootStack.Peek()

    // 3
    Debug.LogFormat("You got a {0}! You've got a good chance of finding
      a {1} next!", currentItem, nextItem);

    Debug.LogFormat("There are {0} random loot items waiting for you!",
      lootStack.Count);
}
```

Происходит вот что.

1. Вызываем метод `Pop` на `lootStack`, удаляем следующий элемент в стеке и сохраняем его.

2. Вызываем метод `Peek` на `lootStack` и сохраняем следующий элемент в стеке, не удаляя его.
3. Выводим в консоль название извлеченного элемента и следующего элемента в стеке.

Вы можете видеть из консоли, что `Mythril Bracers`, последний элемент, добавленный в стек, был удален первым. За ним идет `Winged Boots`, на который мы посмотрели, но не удалили. Вы также можете видеть, что в `lootStack` осталось еще четыре элемента, к которым можно получить доступ (рис. 11.2).

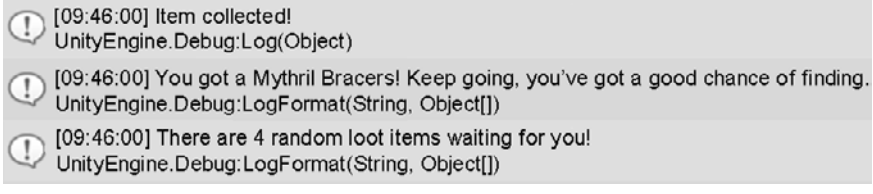


Рис. 11.2

Теперь, когда вы знаете, как создавать, добавлять и запрашивать элементы из стека, мы можем рассмотреть еще несколько методов, предназначенных для работы со стеком.

Общие методы

Для начала вы можете использовать метод `Clear`, чтобы удалить все содержимое стека:

```
// Очистка стека и сброс счетчика на 0
lootStack.Clear()
```

Если хотите узнать, существует ли в вашем стеке определенный элемент, то используйте метод `Contains` и укажите элемент, который ищете:

```
// Возвращает true для элемента "Golden Key"
var itemFound = lootStack.Contains("Golden Key");
```

Если вам нужно скопировать элементы стека в массив, то метод `CopyTo` позволит вам указать место и стартовый индекс для операции копирования:

```
// Копирование элементов стека в массив с индексом, начиная с 0
string[] copiedLoot = new string[lootStack.Count];
numbers.CopyTo(copiedLoot, 0);
```

Если вам нужно преобразовать стек в массив, то используйте метод `ToArray`. После преобразования из вашего стека будет создан новый массив. Отличие от метода `CopyTo` состоит в том, что последний копирует элементы стека в уже существующий массив.

Вы также можете преобразовать стек в строку, если необходимо, используя метод `ToString`:

```
// Копирование существующего стека в новый массив
lootStack.ToArray();

// Преобразование стека в строку
lootStack.ToString();
```

Если вы стараетесь придерживаться хороших практик кодирования, то вам может пригодиться возможность проверить, есть ли в стеке следующий элемент, прежде чем извлекать или просматривать его. К счастью, у класса стека есть два метода для этого конкретного сценария — `TryPeek` и `TryPop`:

```
// Элемент НЕ УДАЛЯЕТСЯ из стека
bool itemPresent = lootStack.TryPeek(out lootItem);
if(itemPresent)
    Debug.Log(lootItem);
else
    Debug.Log("Stack is empty.");

// Элемент УДАЛЯЕТСЯ из стека
bool itemPresent = lootStack.TryPop(out lootItem);
if(itemPresent)
    Debug.Log(lootItem);
else
    Debug.LogFormat("Stack is empty.");
```

Оба метода возвращают `true` или `false` в зависимости от наличия объекта в верхушке стека. Если объект есть, то он будет скопирован в out-параметр результата, и метод вернет `true`. В случае пустого стека значением out-параметра по умолчанию будет его начальное значение, и метод вернет `false`.

Вы можете найти полный список методов стека в документации языка C# по адресу docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.stack-1?view=net-5.0&viewFallbackFrom=netcore-%203.%201.

На этом завершим знакомство со стеками, а в следующем разделе поговорим о схожей структуре — очереди.

Работа с очередями

Как и стеки, очереди — это набор элементов или объектов одного типа. Длина любой очереди переменная, как и у стека, то есть размер изменяется по мере добавления или удаления элементов. Однако очереди работают по модели **first in first out (FIFO)**: первый элемент в очереди является первым доступным элементом. Обратите внимание: в очереди могут храниться повторяющиеся значения и `null`, и при создании ее нельзя сразу заполнить элементами.

Базовый синтаксис

Объявление переменной очереди должно соответствовать следующим требованиям:

- ключевое слово `Queue`, тип элементов очереди в треугольных скобках, а также уникальное имя объекта;
- ключевое слово `new` для инициализации очереди в памяти, за которым следует ключевое слово `Queue` и тип элементов очереди в треугольных скобках;
- пара круглых скобок с точкой с запятой в конце.

В общем виде это выглядит так:

```
Queue<elementType> name = new Queue<elementType>();
```

C# поддерживает необобщенную версию типа Queue, для которой не требуется задавать тип хранимого элемента:

```
Queue myQueue = new Queue();
```

Использовать такой тип менее безопасно и дороже, чем описанную выше версию. Почитайте рекомендации Microsoft на этот счет: github.com/dotnet/platform-compat/blob/master/docs/DE0006.md.

Сама по себе пустая очередь не так уж и полезна, поскольку нам нужна возможность добавлять, удалять и просматривать ее элементы в любой момент, о чем и поговорим в следующем подразделе.

Добавление, удаление и просмотр

Поскольку переменную `lootStack` из предыдущих разделов можно сделать и очередью, мы не будем использовать следующий код в наших игровых сценариях из соображений эффективности. Однако вы можете исследовать различия или сходства этих классов в собственном коде.

Чтобы создать очередь из строковых элементов, напишите следующее:

```
// Создание очереди из строковых элементов
Queue<string> activePlayers = new Queue<string>();
```

Чтобы добавить элементы в очередь, вызовите метод `Enqueue` с элементом, который вы хотите добавить:

```
// Добавление элементов в конец очереди
activePlayers.Enqueue("Harrison");
activePlayers.Enqueue("Alex");
activePlayers.Enqueue("Haley");
```

Чтобы увидеть первый элемент в очереди, не удаляя его, используйте метод `Peek`:

```
// Возвращает первый элемент в очереди, не удаляя его
var firstPlayer = activePlayers.Peek();
```

Чтобы вернуть и удалить первый элемент в очереди, используйте метод `Dequeue`:

```
// Возвращает и удаляет первый элемент в очереди
var firstPlayer = activePlayers.Dequeue();
```

Теперь, когда вы познакомились с основными функциями очереди, не стесняйтесь исследовать более сложные методы, которые есть у класса очереди.

Общие методы

У очередей и стеков почти одинаковые функции, поэтому мы не будем еще раз говорить о них. Полный список методов и свойств можно посмотреть в документации по C#, пройдя по ссылке docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.queue-1?view=netcore-3.1.

Прежде чем завершить главу, рассмотрим тип коллекции HashSet и математические операции, для которых он подходит.

Использование HashSet

Последний тип коллекции, с которым мы познакомимся в данной главе, — HashSet. Эта коллекция значительно отличается от всех прочих, с которыми мы сталкивались. Она не может хранить повторяющиеся значения и не поддерживает сортировку, то есть ее элементы никоим образом не упорядочиваются. Коллекции HashSet — это как словари, но с ключами, а не с парами «ключ — значение». Они позволяют очень быстро выполнять установку значения и поиск элементов, к чему мы вернемся в конце этого раздела, и идеально подходят для ситуаций, когда нам важны сохранение начального порядка и уникальность элементов.

Базовый синтаксис

Объявление переменной HashSet выполняется следующим образом:

- ключевое слово HashSet, тип элементов в треугольных скобках, а также уникальное имя объекта;
- ключевое слово new для инициализации HashSet в памяти, за которым следует ключевое слово HashSet и тип элементов в треугольных скобках;
- пара круглых скобок с точкой с запятой в конце.

В общем виде это выглядит так:

```
HashSet<elementType> name = new HashSet<elementType>();
```

В отличие от стеков и очередей, при объявлении переменной вы можете инициализировать HashSet значениями по умолчанию:

```
HashSet<string> people = new HashSet<string>();

// ИЛИ

HashSet<string> people = new HashSet<string>() { "Joe", "Joan", "Hank"};
```

Чтобы добавить элементы, используйте метод `Add`, указав ему новый элемент:

```
people.Add("Walter");
people.Add("Evelyn");
```

Чтобы удалить элемент, вызовите метод `Remove` и укажите элемент, который вы хотите удалить из `HashSet`:

```
people.Remove("Joe");
```

Все это пока цветочки и на данном этапе вашего путешествия в мир программирования выглядит довольно знакомым. Но при выполнении операций над множествами коллекция `HashSet` действительно выделяется, и именно это будет темой следующего подраздела.

Выполнение операций

Множествами, над которыми будут выполняться операции, являются вызывающий и передаваемый объекты коллекции. Точнее, такие операции служат для изменения элементов вызывающего можества `HashSet` в зависимости от того, какая операция используется. Мы обсудим это более подробно на примере следующего кода, но сначала рассмотрим три основные операции над множествами, которые чаще всего применяются в задачах программирования.

Далее по тексту текущее множество `currentSet` — это `HashSet`, вызывающий метод операции, а указанное множество `specifiedSet` — это переданный параметр метода `HashSet`. Измененное `HashSet` всегда является текущим множеством:

```
currentSet.Operation(specifiedSet);
```

В оставшейся части раздела мы будем работать с тремя основными операциями:

- `UnionWith` складывает элементы текущего и указанного множеств вместе;
- `IntersectWith` возвращает только элементы, которые есть как в текущем, так и в указанном множествах;

- `ExceptWith` вычитает элементы указанного множества из текущего множества.

Существует еще две группы операций над множествами, которые применяются для вычисления подмножеств и надмножеств, но задачи их использования довольно специфичны и выходят за рамки данной главы. Вы можете найти всю необходимую информацию об этих методах, пройдя по ссылке docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.hashset-1?view=net-5.0&viewFallbackFrom=netcore-3.

Допустим, у нас есть два множества с именами игроков — в одном содержатся имена активных игроков, а в другом — неактивных игроков:

```
HashSet<string> activePlayers = new HashSet<string>() { "Harrison",
"Alex", "Haley"};
HashSet<string> inactivePlayers = new HashSet<string>() { "Kelsey",
"Basel"};
```

Мы могли бы использовать операцию `UnionWith`, чтобы получить множество, содержащее элементы обоих исходных множеств:

```
activePlayers.UnionWith(inactivePlayers);
// activePlayers now stores "Harrison", "Alex", "Haley", "Kelsey", "Basel"
```

Теперь предположим, что у нас есть два разных множества — имена активных игроков и имена премиум-игроков:

```
HashSet<string> activePlayers = new HashSet<string>() { "Harrison", "Alex",
"Haley"};
HashSet<string> premiumPlayers = new HashSet<string>() { "Haley", "Basel" };
```

Можно использовать операцию `IntersectWith`, чтобы найти всех активных игроков, которые также являются премиум-игроками:

```
activePlayers.IntersectWith(premiumPlayers);
// Теперь activePlayers хранит только "Haley"
```

Что, если бы нам нужно было получить имена всех активных игроков, не относящихся при этом к премиум-классу? Противоположностью операции `IntersectWith` является `ExceptWith`:

```
HashSet<string> activePlayers = new HashSet<string>() { "Harrison", "Alex",
"Haley"};
HashSet<string> premiumPlayers = new HashSet<string>() { "Haley", "Basel" };

activePlayers.ExceptWith(premiumPlayers);
// Теперь activePlayers хранит "Harrison" и "Alex", но удаляет "Haley"
```

Обратите внимание: я использую новые экземпляры исходных множеств для каждой операции, поскольку текущее множество изменяется после выполнения каждой операции. Если вы продолжите использовать одни и те же множества, то результаты могут оказаться неожиданными.

Теперь, когда вы узнали, как выполнять быстрые математические операции с помощью `HashSet`, закончим эту главу и вспомним все, о чем говорили.

Подведем итоги

Поздравляю, вы почти на финишной прямой! В данной главе мы изучили три новых типа коллекций и их использование в различных ситуациях. Стеки отлично подходят для ситуаций, когда нужно получить доступ к элементам коллекции в порядке, противоположном порядку добавления. Очереди — ваш кандидат, если вы хотите получить доступ к своим элементам в порядке добавления, и оба варианта идеально подходят для временного хранения. Важное различие между этими типами коллекций и списками или массивами заключается в том, как обеспечивается доступ к ним с помощью операций извлечения и просмотра. Наконец, вы узнали о `HashSet` и математических операциях над множествами, которые выполняются невероятно быстро. Когда нужно работать с уникальными значениями и выполнять сложение, сравнение или вычитание больших коллекций, они подходят просто прекрасно.

В следующей главе мы еще сильнее погрузимся в более сложные функции C# и познакомимся с делегатами, обобщениями (generics; иногда называют дженериками) и др., приближаясь к концу книги. Но даже последняя ее страница станет лишь началом нового пути.

Контрольные вопросы. Сложные коллекции

1. В какой коллекции элементы хранятся с использованием модели LIFO?
2. Какой метод позволяет запрашивать следующий элемент в стеке, не удаляя его?
3. Можно ли хранить в стеках и очередях значения `null`?
4. Как вычесть один `HashSet` из другого?

12 Обобщения, делегаты и многое другое

Чем больше времени вы проводите за программированием, тем больше думаете о системах. Структурирование механизмов, по которым классы и объекты взаимодействуют и обмениваются друг с другом данными, — это и есть то, чем мы занимались до сих пор. Теперь вопрос в том, как сделать их более безопасными и эффективными.

Это будет последняя практическая глава книги, и в ней мы рассмотрим примеры обобщенных концепций программирования, делегирования, создания событий и обработки ошибок. Каждая из этих тем сама по себе является обширной областью, достойной изучения, поэтому возьмите все, что мы изучим здесь, и развивайте эти знания в своих проектах. Закончив заниматься практикой кодирования, мы приведем небольшой обзор паттернов проектирования и того, какова их роль в вашем будущем программировании.

В этой главе мы рассмотрим такие темы, как:

- обобщенное программирование;
- использование делегатов;
- создание событий и подписок;
- выдача и обработка ошибок;
- обзор паттернов проектирования.

Обобщения

Весь код, который мы писали, был очень специфичным с точки зрения определения и использования типов. Однако будут случаи, когда вам понадобится класс или метод для аналогичной обработки разных

сущностей, независимо от их типа, но с сохранением типобезопасности. Обобщенное программирование позволяет нам создавать более универсальные классы, методы и переменные, используя заполнители, а не конкретный тип.

Когда создается конкретный экземпляр обобщенного класса или вызывается обобщенный метод, ему присваивается конкретный тип, но сам код ориентирован на обобщенный тип. Чаще всего обобщенное программирование встречается при работе с пользовательскими типами коллекций, которые должны выполнять одни и те же операции с элементами независимо от их типа. Возможно, сейчас все сказанное не имеет для вас смысла, но, когда мы рассмотрим конкретный пример ниже, все встанет на свои места.

Мы уже видели, как это работает, на примере типа List, который сам по себе является обобщенным типом. Мы можем использовать методы добавления, удаления и изменения элементов независимо от того, хранятся в списке целые числа, строки или отдельные символы.

Обобщенные объекты

Обобщенный класс создается так же, как обычный, но с одним важным отличием: у него должен быть параметр обобщенного типа. Рассмотрим пример обобщенного класса коллекции, который пришло время создать, чтобы получить более четкое представление о том, как это работает:

```
public class SomeGenericCollection<T> {}
```

Мы объявили обобщенный класс с именем `SomeGenericCollection` и указали, что его параметр типа будет называться `T`. Теперь вместо `T` будет использоваться тип элемента, который будет храниться в обобщенном списке, и этот тип может использоваться внутри обобщенного класса так же, как и любой другой тип.

Всякий раз, когда мы создаем экземпляр `SomeGenericCollection`, нам нужно указывать тип значений, которые он может хранить:

```
SomeGenericCollection<int> highScores = new SomeGenericCollection<int>
  ();
```

В этом случае в `highScores` хранятся целочисленные значения, а `T` обозначает тип `int`, но класс `SomeGenericCollection` будет работать с любыми элементами одинаково.

Вы можете задавать параметру обобщенного типа любое имя, но отраслевой стандарт во многих языках программирования — это именование с заглавной буквы T. Если вы собираетесь называть параметры своего типа по-другому, то рассмотрите опцию начинать их имена с T ради согласованности и удобочитаемости.

Время действовать. Создаем обобщенную коллекцию

Создадим более полный обобщенный класс `list` для хранения некоторых предметов в инвентаре, выполнив следующие действия.

1. Создайте новый сценарий C# в папке Scripts, назовите его `InventoryList` и добавьте в него следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// 1
public class InventoryList<T>
{
    // 2
    public InventoryList()
    {
        Debug.Log("Generic list initalized...");
    }
}
```

2. Создайте новый экземпляр `InventoryList` в сценарии `GameBehavior`:

```
public class GameBehavior : MonoBehaviour, IManager
{
    // ... Изменения не требуются ...

    void Start()
    {
        Initialize();

        // 3
        InventoryList<string> inventoryList =
```

```
            new InventoryList<string>();
        }

        // ... Нет изменений в Initialize или OnGUI ...
    }
```

Разберем этот код.

1. Объявляем новый обобщенный класс с именем `InventoryList` и типом параметра `T`.
2. Добавляем конструктор по умолчанию с простым выводом в консоль при появлении нового экземпляра `InventoryList`.
3. Создаем новый экземпляр `InventoryList` в сценарии `GameBehavior` для хранения строковых значений (рис. 12.1).

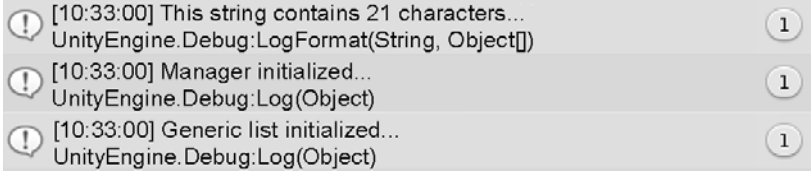


Рис. 12.1

С точки зрения функциональности ничего нового тут пока не произошло, но Visual Studio распознает `InventoryList` как обобщенный класс из-за его параметра обобщенного типа `T`. Это позволяет нам добавлять дополнительные обобщенные операции в сам класс `InventoryList`.

Обобщенные методы

У автономного обобщенного метода, как и у обобщенного класса, должен быть заполнитель типа параметра, что позволяет при необходимости включать такие методы как в обобщенный, так и в необобщенный класс:

```
public void GenericMethod<T>(T genericParameter) {}
```

Тип `T` может использоваться внутри тела метода и определяться при вызове метода:

```
GenericMethod<string>("Hello World!");
```

Однако если вы хотите объявить обобщенный метод внутри обобщенного класса, то не нужно указывать еще один тип T:

```
public class SomeGenericCollection<T>
{
    public void NonGenericMethod(T genericParameter) {}
}
```

Когда вы вызываете необобщенный метод с параметром обобщенного типа, проблем нет, поскольку обобщенный класс сам назначит конкретный тип вместо обобщенного:

```
SomeGenericCollection<int> highScores = new SomeGenericCollection
                                                        <int> ();
highScores.NonGenericMethod(35);
```

Подобно необобщенным методам, обобщенные можно перегружать и помечать как статические. Если вам нужна справка по синтаксису для этих ситуаций, то перейдите по ссылке docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/generics/generic-methods.

Ваша следующая задача — создать новый обобщенный элемент и использовать его в сценарии `InventoryList`.

Время действовать. Добавляем обобщенный предмет

У нас уже есть обобщенный класс с определенным типом параметра, поэтому добавим необобщенный метод, чтобы посмотреть, как они будут работать вместе.

1. Откройте сценарий `InventoryList` и добавьте в него код:

```
public class InventoryList<T>
 {
     // 1
     private T _item;
     public T item
     {
         get { return _item; }
     }
```

```
    public InventoryList()
    {
        Debug.Log("Generic list initialized...");
    }

    // 2
    public void SetItem(T newItem)
    {
        // 3
        _item = newItem;
        Debug.Log("New item added...");
    }
}
```

2. Откройте сценарий `GameBehavior` и добавьте элемент в `InventoryList`:

```
public class GameBehavior : MonoBehaviour, IManager
{
    // ... Изменения не требуются ...

    void Start()
    {
        Initialize();
        InventoryList<string> inventoryList =
        new InventoryList<string>();

        // 4
        inventoryList.SetItem("Potion");
        Debug.Log(inventoryList.item);
    }

    public void Initialize()
    {
        // ... Изменения не требуются ...
    }

    void OnGUI()
    {
        // ... Изменения не требуются ...
    }
}
```

Разберем этот код.

1. Добавляем публичный параметр `item` типа `T` с приватной резервной переменной того же типа.

2. Объявляем новый метод внутри `InventoryList`, назовем его `SetItem`, который принимает параметр типа `T`.
3. Задаем значение `_item` обобщенному параметру, переданному в `SetItem()`, и выводим в консоль сообщение об успехе.
4. Присваиваем строковое значение параметру `item` в `InventoryList` с помощью метода `SetItem()` и выводим сообщение в консоль (рис. 12.2).

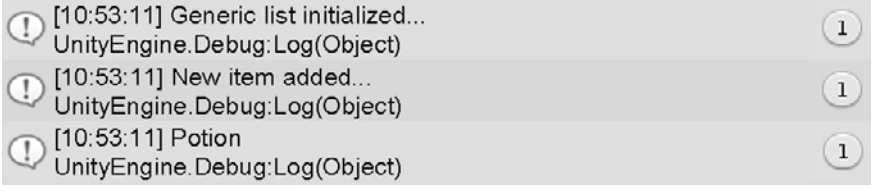


Рис. 12.2

Мы научили метод `SetItem()` принимать параметр любого типа, с которым создан наш общий экземпляр `InventoryList`, и назначать его новому свойству класса, используя публичный и приватный модификаторы. Поскольку `InventoryList` был создан для хранения строковых значений, мы без проблем назначили свойству `item` строку `"Potion"`. Однако этот метод одинаково хорошо работает с любым типом, с которым может быть создан экземпляр `InventoryList`.

Ограничения типов параметров

Одна из замечательных особенностей обобщений заключается в том, что типы их параметров можно ограничить. Это может показаться нелогичным в свете полученных знаний об обобщениях, но тот факт, что класс может содержать любой тип, не означает, что это всегда хорошо.

Чтобы ограничить возможные типы параметров у обобщенного класса, нам нужны новое ключевое слово и новый синтаксис:

```
public class SomeGenericCollection<T> where T: ConstraintType {}
```

Ключевое слово `where` определяет правила, которым должен отвечать тип `T`, прежде чем его можно будет использовать в качестве параметра

обобщенного типа. По сути, мы говорим, что `SomeGenericClass` может принимать любой тип `T`, если он соответствует ограничивающему типу. В ограничивающих правилах нет ничего мистического или пугающего, и здесь используются уже известные нам концепции:

- добавление ключевого слова `class` ограничит `T` типами, которые являются классами;
- добавление ключевого слова `struct` ограничит `T` типами, которые являются структурами;
- добавление интерфейса, такого как `IManager`, в качестве типа ограничит `T` типами, которые реализуют этот интерфейс;
- добавление пользовательского класса, такого как `Character`, ограничит `T` только этим типом класса.

Если вам нужен более гибкий подход для поддержки классов, у которых есть подклассы, то вы можете использовать синтаксис where T: U, указывающий, что обобщенный тип T должен либо совпадать с типом U, либо наследоваться от него. Для наших задач это уже лишнее, но вы можете найти более подробную информацию на docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/generics/constraints-on-type-parameters.

Время действовать. Ограничиваем обобщенные элементы

Просто для иллюстрации ограничим класс `InventoryList`, чтобы он мог принимать только те типы, которые являются классами.

Откройте `InventoryList` и добавьте следующий код:

```
// 1
public class InventoryList<T> where T: class
{
    // ... Изменения не требуются ...
}
```

Поскольку наш экземпляр `InventoryList` работает со строками, а те являются классами, проблем в работе кода нет. Однако если бы в качестве ограничения типа указали структуру или имя интерфейса, то наш обобщенный класс выдал бы ошибку. Эта возможность полезна, когда

вам нужно защитить свои обобщенные классы или методы от типов, которые вы не хотите поддерживать.

Делегирование действий

Бывают моменты, когда вам нужно передать, или *делегировать*, фактическое выполнение метода другому объекту. В C# это можно сделать с помощью делегатов, которые хранят ссылки на методы и могут обрабатываться как любая другая переменная. Единственный нюанс заключается в том, что сам делегат и любой назначенный метод должны иметь одну и ту же сигнатуру. Аналогично целочисленные переменные могут содержать только целые числа, а строки — лишь текст.

Базовый синтаксис

Создание делегата — это смесь написания функций и объявления переменных:

```
public delegate returnType DelegateName(int param1, string param2);
```

Сначала указываем модификатор доступа, а за ним следует ключевое слово `delegate`, которое идентифицирует его для компилятора как тип `delegate`. Данный тип может иметь возвращаемый тип и имя, как и любая другая функция, а также параметры, если нужно. Однако этот синтаксис объявляет только сам тип `delegate`. Чтобы использовать его, вам нужно создать экземпляр, как и при работе с классами:

```
public DelegateName someDelegate;
```

Имея объявленный тип переменной `delegate`, легко назначить метод, соответствующий сигнатуре делегата:

```
public DelegateName someDelegate = MatchingMethod;

public void MatchingMethod(int param1, string param2)
{
    // ... Строки кода ...
}
```

Обратите внимание: мы не используем круглые скобки при назначении `MatchingMethod` переменной `someDelegate`, поскольку на данном этапе метод не вызывается. Ответственность за вызов `MatchingMethod` передается `someDelegate`, из чего следует, что мы можем вызывать функцию так:

```
someDelegate();
```

На данном этапе этот навык может показаться обременительным на вашем и без того сложном пути в изучении C#, но я обещаю вам, что возможность хранить и выполнять методы как переменные пригодится в будущем.

Время действовать. Создаем делегат отладки

Создадим простой делегат, определим метод, который принимает строку и выводит ее с помощью назначенного метода.

Откройте сценарий `GameBehavior` и добавьте следующий код:

```
public class GameBehavior : MonoBehaviour, IManager
 {
     // ... Другие изменения не требуются ...

     // 1
     public delegate void DebugDelegate(string newText);

     // 2
     public DebugDelegate debug = Print;

     // ... Другие изменения не требуются ...

     void Start()
     {
         // ... Изменения не требуются ...
     }

     public void Initialize()
     {
         _state = "Manager initialized..";
         _state.FancyDebug();

         // 3
         debug(_state);
     }

     // 4
     public static void Print(string newText)
```

```
    {
        Debug.Log(newText);
    }

    void OnGUI()
    {
        // ... Изменения не требуются ...
    }
}
```

Разберем этот код.

1. Объявляем `public delegate` с именем `DebugDelegate` и в нем будем хранить метод, который требует параметр `string` и возвращает `void`.
2. Создаем новый экземпляр `DebugDelegate` под именем `debug` и назначаем ему метод с соответствующей сигнатурой и именем `Print()`.
3. Заменим `Debug.Log(_state)` в методе `Initialize()` вызовом переменной `debug`.
4. Объявляем `Print()` как статический метод, который принимает параметр `string` и записывает его в консоль (рис. 12.3).

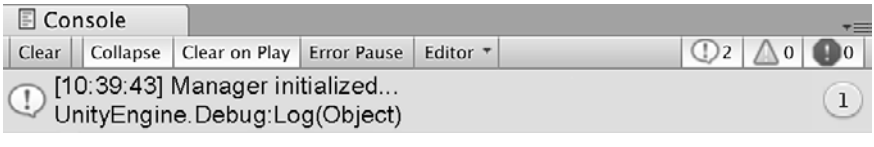


Рис. 12.3

В консоли ничего не изменилось, но вместо прямого вызова `Debug.Log()` внутри метода `Initialize()` мы делегировали те же действия экземпляру делегата `debug`. Это весьма упрощенный пример, но в целом делегирование — эффективный инструмент, когда вам нужно хранить, передавать и выполнять методы как их типы. Мы уже работали с реальным примером делегирования с использованием методов `OnCollisionEnter()` и `OnCollisionExit()`, поскольку эти методы Unity вызываются с помощью делегирования.

Делегаты как типы параметров

Поскольку мы уже попробовали создавать типы делегатов для хранения методов, имеет смысл использовать тип делегата в качестве самого

параметра метода. Это не слишком отличается от того, что мы уже делали, но неплохо было бы охватить и данный вопрос.

Время действовать. Используем аргумент делегата

Посмотрим, как тип делегата можно использовать в качестве параметра метода.

Добавьте в `GameBehavior` следующий код:

```
public class GameBehavior : MonoBehaviour, IManager
 {
     // ... Изменения не требуются ...

     void Start()
     {
         // ... Изменения не требуются ...
     }

     public void Initialize()
     {
         _state = "Manager initialized..";
         _state.FancyDebug();

         debug(_state);

         // 1
         LogWithDelegate(debug);
     }

     public static void Print(string newText)
     {
         // ... Изменения не требуются ...
     }

     // 2
     public void LogWithDelegate(DebugDelegate del)
     {
         // 3
         del("Delegating the debug task...");
     }

     void OnGUI()
     {
         // ... Изменения не требуются ...
     }
 }
```

Разберем этот код.

1. Вызываем метод `LogWithDelegate()` и передаем переменную `debug` как параметр.
2. Объявляем новый метод, который принимает параметр `DebugDelegate`.
3. Вызываем функцию параметра делегата и передаем строковый литерал для вывода в консоль (рис. 12.4).

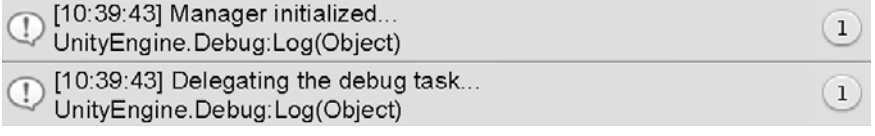


Рис. 12.4

Мы создали метод, который принимает параметр типа `DebugDelegate`; это означает, что фактический переданный аргумент будет представлять метод и может рассматриваться соответственно. Мы создали своего рода цепочку делегирования, где метод `LogWithDelegate()` отстоит на два шага от фактического метода, выполняющего вывод в консоль, то есть `Print()`.

В цепочках делегирования легко потеряться. Если вы потеряете, что за чем идет, то вернитесь и просмотрите код с начала раздела и проверьте документацию на docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/delegates/.

Теперь, когда вы знаете, как работать с основными делегатами, пора поговорить о событиях и о том, как их можно использовать для эффективного обмена информацией между сценариями.

Запуск событий

События C# позволяют вам создать систему подписок, основанную на действиях, выполняемых в ваших играх или приложениях. Например, если вы хотите запускать некое событие всякий раз, когда подбирается

предмет или когда игрок нажимает пробел, сделать это можно. Однако по умолчанию при возникновении события у него не будет подписчика или получателя для обработки любого кода, который необходимо выполнить после возникновения события.

Любой класс может подписаться на событие или отказаться от подписки через вызывающий класс, из которого оно запускается. Подобно подписке на уведомления от сообщества в Facebook, события образуют своего рода супермагистраль распределенной информации для обмена действиями и данными в вашем приложении.

Базовый синтаксис

Объявление событий в чем-то аналогично объявлению делегатов. У события тоже должна быть определенная сигнатура метода. Мы применим делегат, чтобы указать сигнатуру метода, которая требуется нам для события, а затем создадим событие, используя тип `delegate` и ключевое слово `event`:

```
public delegate void EventDelegate(int param1, string param2);
public event EventDelegate eventInstance;
```

Этот код позволяет нам рассматривать `eventInstance` как метод, поскольку мы назначили ему тип делегата, и в результате мы сможем активировать событие в любое время, вызвав его:

```
eventInstance(35, "John Doe");
```

Ваша следующая задача — создать собственное событие и запустить его в нужном месте в сценарии `PlayerBehavior`.

Время действовать. Создаем событие

Создадим событие, которое будет запускаться во время каждого прыжка игрока.

Откройте `PlayerBehavior` и внесите в него следующие изменения:

```
public class PlayerBehavior : MonoBehaviour
 {
     // ... Изменения переменных не требуются...
```

```
    // 1
    public delegate void JumpingEvent();

    // 2
    public event JumpingEvent playerJump;

    void Start()
    {
        // ... Изменения не требуются ...
    }

    void Update()
    {
        _vInput = Input.GetAxis("Vertical") * moveSpeed;
        _hInput = Input.GetAxis("Horizontal") * rotateSpeed;
    }

    void FixedUpdate()
    {
        if(IsGrounded() && Input.GetKeyDown(KeyCode.Space))
        {
            _rb.AddForce(Vector3.up * jumpVelocity,
            ForceMode.Impulse);

            // 3
            playerJump();
        }
    }

    // ... Изменения в IsGrounded или OnCollisionEnter не требуются
}
```

Разберем этот код.

1. Объявляем новый тип `delegate`, который возвращает `void` и не принимает никаких параметров.
2. Создаем событие типа `JumpingEvent` с названием `playerJump`. Событие можно рассматривать как метод, который соответствует возвращаемому делегатом `void` и без сигнатуры параметра.
3. Вызываем `playerJump` после приложения силы в методе `Update()`.

Мы успешно создали простой тип делегата, который не принимает параметров и ничего не возвращает, а также событие того же типа, выполняемое всякий раз, когда игрок прыгает. После каждого прыжка

событие `playerJump` отправляется всем его подписчикам, уведомляя их о том, что прыжок произошел.

После запуска события подписчики должны обработать его и выполнить соответствующие операции — об этом в следующем подразделе.

Обработка подписок на события

В данный момент у нашего события `playerJump` нет подписчиков, но это легко исправить, подобно тому как мы назначали ссылки на методы для типов делегатов выше:

```
someClass.eventInstance += EventHandler;
```

Поскольку события — это переменные, принадлежащие тому классу, в котором они объявлены, а подписчиками будут другие классы, для «оформления подписки» необходима ссылка на класс, содержащий событие. Оператор `+=` используется для назначения метода, который будет срабатывать при выполнении события (это похоже на рассылки по почте с работы). Как и при назначении делегатов, сигнатура метода обработчика события должна соответствовать типу события. В предыдущем примере синтаксиса это означает, что обработчик `EventHandler` должен выглядеть как-то так:

```
public void EventHandler(int param1, string param2) {}
```

В случаях, когда вам нужно отказаться от подписки на событие, можно использовать оператор `-=`:

```
someClass.eventInstance -= EventHandler;
```

Подписки на события обычно обрабатываются при инициализации или уничтожении экземпляра класса, что упрощает управление несколькими событиями и позволяет не писать много беспорядочного кода.

Теперь, когда вы знаете синтаксис для подписки и отказа от подписки на события, настала ваша очередь применить это на практике в сценарии `GameBehavior`.

Время действовать. Подписываемся на событие

Теперь, когда наше событие запускается каждый раз, когда игрок прыгает, нам нужен способ зафиксировать это действие.

Вернитесь к сценарию `GameBehavior` и обновите следующий код:

```
public class GameBehavior : MonoBehaviour, IManager
 {
     // ... Изменения не требуются ...

     void Start()
     {
         // ... Изменения не требуются ...
     }

     public void Initialize()
     {
         _state = "Manager initialized..";
         _state.FancyDebug();

         debug(_state);
         LogWithDelegate(debug);

         // 1
         GameObject player = GameObject.Find("Player");

         // 2
         PlayerBehavior playerBehavior =
         player.GetComponent<PlayerBehavior>();

         // 3
         playerBehavior.playerJump += HandlePlayerJump;
     }

     // 4
     public void HandlePlayerJump()
     {
         debug("Player has jumped...");
     }

     // ... Нет изменений в Print,
            LogWithDelegate или
            OnGUI ...
 }
```

Разберем этот код.

1. Находим объект `Player` в сцене и сохраняем его `GameObject` в локальной переменной.
2. С помощью метода `GetComponent()` получаем ссылку на класс `PlayerBehavior`, прикрепленный к `Player`, и сохраняем его в локальной переменной.
3. Подписываемся на событие `PlayerJump`, объявленное в классе `PlayerBehavior` с помощью метода `HandlePlayerJump`.
4. Объявляем метод `HandlePlayerJump()` с сигнатурой, соответствующей типу события, и регистрируем сообщение об успешном завершении с использованием делегата `debug` каждый раз, когда получаем событие (рис. 12.5).

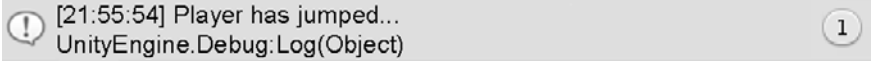


Рис. 12.5

Чтобы правильно подписаться на события в `GameBehavior`, нам нужно было получить ссылку на класс `PlayerBehavior`, прикрепленный к игроку. Мы могли бы сделать все это в одной строке, но в отдельных строках будет намного удобнее читать код. Затем мы присвоили событию `PlayerJump` метод, который будет выполняться всякий раз при получении события, и завершили процесс подписки. Поскольку в нашем прототипе игрок никогда не уничтожается, нет необходимости отменять подписку на `PlayerJump`, однако не забудьте сделать этот шаг, если того требует ситуация.

На этом мы завершаем разговор о событиях, и нам осталось обсудить очень важную тему, без которой не может добиться успеха ни одна программа, — обработку ошибок.

Обработка исключений

Эффективная обработка ошибок и исключений в коде иллюстрирует ваш уровень профессионализма и владения программированием. Прежде чем вы начнете восклицать: «Зачем мне самому писать

ошибки, если я только и делал, что пытался избежать их?!» — нужно понять, что я не имею в виду добавление ошибок для нарушения работы существующего кода. Совсем наоборот — включение ошибок или исключений и правильная обработка ситуаций некорректного использования функций кода повышают устойчивость вашей кодовой базы к сбоям.

Выбрасывание исключений

Когда мы говорим о добавлении ошибок, мы называем данный процесс *выбрасыванием исключения*. Это часть так называемого защитного программирования, что по сути означает активную и сознательную защиту кода от неправильных или незапланированных операций. Чтобы отметить такие ситуации, вы выбрасываете исключение из метода, которое затем обрабатывается вызывающим кодом.

Приведу пример: допустим, у нас есть оператор `if`, который проверяет, правильно ли игрок ввел свой адрес электронной почты во время регистрации. Если адрес электронной почты введен некорректно, то код должен выбрасывать исключение:

```
public void ValidateEmail(string email)
{
    if(!email.Contains("@"))
    {
        throw new System.ArgumentException("Email is invalid");
    }
}
```

Для выбрасывания исключения мы используем ключевое слово `throw`. Исключение создается с помощью ключевого слова `new`, за которым следует непосредственно исключение. Конструктор `System.ArgumentException()` по умолчанию выдает информацию о том, где и когда исключение было выполнено, но может и принимать пользовательскую строку, если вы хотите переделать его вывод.

`ArgumentException` — это подкласс класса `Exception`, доступ к которому осуществляется через класс `System`. В C# уже заложено множество встроенных типов исключений, но мы не будем углубляться в них, поскольку это вы сможете сделать и сами, поняв общую идею системы.

Полный список исключений языка C# можно найти в разделе «Выбор стандартных исключений» на docs.microsoft.com/ru-ru/dotnet/api/system.exception?view=netframework-4.7.2#Standard.

Время действовать. Проверяем отрицательные индексы сцены

Познакомимся с исключениями и убедимся, что наш уровень перезапускается только в том случае, если переданный методу номер сцены положителен.

1. Откройте класс `Utilities` и добавьте следующий код в перегруженную версию метода `RestartLevel()`:

```
public static class Utilities
 {
     public static int playerDeaths = 0;

     public static string UpdateDeathCount(out int countReference)
     {
         // ... Изменения не требуются ...
     }

     public static void RestartLevel()
     {
         // ... Изменения не требуются ...
     }

     public static bool RestartLevel(int sceneIndex)
     {
         // 1
         if(sceneIndex < 0)
         {
             // 2
             throw new System.ArgumentException("Scene index
             cannot be negative");
         }

         SceneManager.LoadScene(sceneIndex);
         Time.timeScale = 1.0f;

         return true;
     }
 }
```

2. В вызове метода `RestartLevel()` в методе `OnGUI()` сценария `GameBehavior` передайте отрицательный индекс сцены и поиграйте в игру, пока не проиграете:

```
if(showLossScreen)
{
      if (GUI.Button(new Rect(Screen.width / 2 - 100,
        Screen.height / 2 - 50, 200, 100), "You lose..."))
      {
           // 3
           Utilities.RestartLevel(-1);
      }
}
```

Разберем этот код.

1. Объявляем оператор `if`, который проверяет `sceneIndex` на предмет положительности.
2. Выбрасываем исключение `ArgumentException` с заданным сообщением, если в качестве аргумента передается отрицательный индекс сцены.
3. Вызываем метод `RestartLevel()` с индексом сцены `-1` (рис. 12.6).

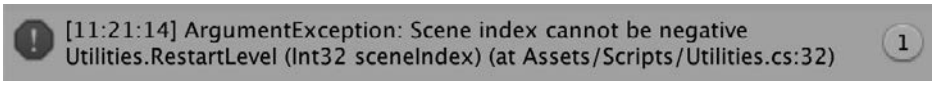


Рис. 12.6

Когда мы проигрываем игру, вызывается метод `RestartLevel()`, но, поскольку мы передали в качестве индекса сцены значение `-1`, наше исключение выполнится до того, как заработает диспетчер сцены. Игра остановится, так как на данный момент у нас нет других вариантов развития событий, но главное, что защита работает ожидаемым образом и не позволяет нам предпринимать действия, которые могут привести к сбою игры (Unity не поддерживает отрицательные индексы сцен).

Теперь, когда вы успешно сгенерировали ошибку, посмотрим, как работать с последствиями ошибок, — в следующем подразделе рассмотрим оператор `try-catch`.

Использование оператора try-catch

На текущий момент наша задача — безопасно обработать возможные результаты, которые может иметь вызов `RestartLevel()`, поскольку на данном этапе это не решается должным образом. Здесь на помощь придет оператор нового типа, называемый `try-catch`:

```
try
{
    // Вызов метода, который может выбросить исключение
}
catch (ExceptionType localVariable)
{
    // Перехват всех исключений по отдельности
}
```

Оператор `try-catch` состоит из нескольких блоков кода, которые выполняются в разных условиях. В чем-то он похож на несколько видоизмененный оператор `if / else`. В блоке `try` мы вызываем любые методы, которые потенциально могут выбрасывать исключения. Если исключение не генерируется, то код продолжает работать штатным порядком. Если генерируется, то код переходит к оператору `catch`, который выполняется с учетом данного исключения. Это работает точно так же, как перебор вариантов в операторе `switch`. Операторы `catch` позволяют задать конкретное перехватываемое исключение и указать имя локальной переменной, которая представит собой это исключение внутри блока `catch`.

Вы можете использовать сколько угодно операторов catch после блока try, если нужно обработать несколько исключений, которые могут быть выброшены одним методом.

Кроме того, существует необязательный блок `finally`, который может быть объявлен после любых операторов `catch`. Этот блок будет выполняться в самом конце оператора `try-catch`, независимо от того, было ли выброшено исключение:

```
finally
{
    // Код, который выполняется в конце попытки перехвата
    // независимо от ситуации
}
```

Ваша следующая задача — использовать оператор `try-catch` для обработки любых ошибок, которые возникают при неудачном перезапуске уровня.

Время действовать. Обнаруживаем ошибки перезапуска

Теперь, когда при проигрыше игры у нас выбрасывается исключение, безопасно обработаем его.

Добавьте в сценарий `GameBehavior` следующий код и поиграйте в игру, пока не проиграете:

```
public class GameBehavior : MonoBehaviour, IManager
 {
     // ... Изменения переменных не требуются ...

     // ... Изменения в Start
            Initialize,
            Print
            или LogWithDelegate не требуются...

     void OnGUI()
     {
         // ... Изменения в OnGUI не требуются...

         if(showLossScreen)
         {
             if (GUI.Button(new Rect(Screen.width / 2 - 100,
             Screen.height / 2 - 50, 200, 100), "You lose..."))
             {
                 // 1
                 try
                 {
                     Utilities.RestartLevel(-1);
                     debug("Level restarted successfully...");
                 }
                     // 2
                     catch (System.ArgumentException e)
                 {
                     // 3
                     Utilities.RestartLevel(0);
                     debug("Reverting to scene 0: " + e.ToString());
                 }
                 // 4
                 finally
```

```
                {
                    debug("Restart handled...");
                }
            }
        }
    }
}
```

Разберем этот код.

1. Объявляем блок `try` и перемещаем вызов метода `RestartLevel()` в него, добавив команду `debug`, чтобы вывести информацию о том, произошел ли перезапуск без каких-либо исключений.
2. Объявляем блок `catch` и определяем `System.ArgumentException` как тип исключения, которое будет храниться в локальной переменной.
3. Перезапускаем игру с индексом сцены по умолчанию, если выбрасывается исключение:
   - используем делегат `debug` для вывода своего сообщения, а также информации об исключении, к которой можно получить доступ из переменной `e`, преобразовав ее в строку с помощью метода `ToString()`.

Поскольку переменная e относится к типу ArgumentException, у вас есть несколько свойств и методов, связанных с классом Exception, к которым вы можете получить доступ. Это часто бывает полезно, когда требуется подробная информация о конкретном исключении.

4. Добавляем блок `finally` с сообщением в консоли о том, что обработка исключения выполнена (рис. 12.7).

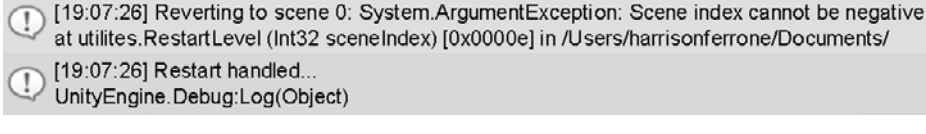


Рис. 12.7

Теперь при вызове функции `RestartLevel()` наш блок `try` безопасно разрешает выполнение, и если возникает ошибка, то код попадает в блок `catch`. Последний перезапускает уровень с индексом сцены по

умолчанию, и код переходит к блоку `finally`, который просто выводит в консоль сообщение.

Механизмы работы с исключениями важно понимать, но не стоит доводить любовь к ним до привычки использовать их повсеместно. Это приведет к раздуванию классов и может повлиять на время обработки игры. Нужно применять исключения там, где они больше всего нужны, учитывая возможный неправильный ввод или обработку данных, а не игровую механику.

C# позволяет вам создавать типы исключений под любые конкретные потребности вашего кода, но эта тема выходит за рамки данной книги. Но если потребуется на будущее, то вам сюда: docs.microsoft.com/ru-ru/dotnet/standard/exceptions/how-to-create-user-defined-exceptions.

Прежде чем мы закончим главу, рассмотрим последнюю тему, с которой вам нужно познакомиться, — паттерны проектирования. Мы не будем вдаваться в написание кода для них (об этом есть целые отдельные книги), но поговорим об их назначении и удобстве использования в программировании.

Экскурс по паттернам проектирования

Прежде чем мы завершим эту главу, нужно поговорить о теме, которая будет играть огромную роль в вашей карьере программиста, если вы решите продолжить заниматься программированием: паттерны проектирования. Поискав в Google паттерны проектирования или паттерны программирования программного обеспечения, вы найдете множество определений и примеров, которые при первом знакомстве лишь обескуражат вас. Упрощу сам термин «паттерн проектирования» и изложу его следующим образом.

Паттерн для решения программных проблем или ситуаций, с которыми вы будете регулярно сталкиваться при разработке любого вида

приложений. Это не жестко запрограммированные решения, а скорее проверенные и показавшие себя рекомендации и передовые методы, которые можно адаптировать под конкретную ситуацию.

Паттерны проектирования стали неотъемлемой частью программирования, и история их становления весьма интересна (покопайтесь в этом сами). Если данная концепция вам интересна, то начните с книги *«Паттерны объектно-ориентированного проектирования»* и ее авторов, *«Банды четырех»*.

Паттерны проектирования, о которых мы говорим, применимы только к языкам и парадигмам объектно-ориентированного программирования (ООП), поэтому у вас не получится их применить, если вы работаете не в ООП.

Часто используемые игровые паттерны

Существует более 35 задокументированных паттернов проектирования, разбитых на четыре категории в зависимости от функциональности, но лишь некоторые из них однозначно подходят для разработки игр. Я приведу краткий обзор паттернов, чтобы вы поняли, о чем идет речь.

- Паттерн `Singleton` гарантирует, что у некоего класса будет только один экземпляр в программе и одна глобальная точка доступа (такой паттерн полезен для классов наподобие менеджера игры).
- Паттерн `Observer` представляет сложную систему уведомлений, которая оповещает подписчиков об изменениях в поведении с помощью событий. Мы видели работу этого паттерна в примерах с делегатами/событиями, но его можно расширить до бóльших масштабов.
- Паттерн `State` позволяет объекту изменять свое поведение в зависимости от состояния. Паттерн полезен для создания умных противников, которые бы меняли тактику в зависимости от действий игрока или условий окружающей среды.
- Паттерн `Object Pool` повторно использует «отработанные» объекты, а не создает новые всякий раз. Этот паттерн отлично подошел бы для

механики стрельбы в игре Hero Born, поскольку игра может начать тормозить, если на машине с низкой вычислительной мощностью будет создано слишком много пуль.

Если мне не удалось убедить вас в том, насколько важны паттерны проектирования, то знайте, что Unity работает на паттерне `Composite` (иногда называемом `Component`), который позволяет нам создавать сложные объекты, состоящие из отдельных функциональных частей.

Опять же это лишь малая часть того, что паттерны проектирования позволяют делать в реальных ситуациях программирования. Я настоятельно рекомендую вам покопаться в их истории и применении, как только вы закончите следующую главу. Уверяю, в будущем они станут вашими лучшими друзьями.

Подведем итоги

В этой главе мы подошли к концу нашего приключения в мире C# и Unity 2020, но я надеюсь, что ваше собственное путешествие в программирование игр и разработку программного обеспечения только начинается. Мы изучили буквально все: от создания переменных, методов и объектов класса до написания игровой механики, поведения врагов и многое другое.

Темы, которые мы рассмотрели в данной главе, были на порядок сложнее того, о чем мы говорили в большей части книги, и тому была причина. Вы уже знаете, что ваш мозг программиста — это мышца, которую нужно тренировать, прежде чем покорять новые вершины. Обобщения, события и паттерны проектирования — та самая следующая ступень на вашей лестнице программирования.

В следующей главе я расскажу вам, что делать дальше: покажу ресурсы, дополнительную информацию и множество других полезных (и, осмелюсь сказать, замечательных) возможностей и сведений о сообществе Unity и индустрии разработки программного обеспечения в целом.

Рабочего кода!

Контрольные вопросы. Продвинутый C#

1. В чем разница между обобщенным и необобщенным классом?
2. Что должно совпадать при присвоении значения типу делегата?
3. Как отписать объект от события?
4. Какое ключевое слово C# позволяет выбросить исключение в коде?

13 Путешествие продолжается

Если вы открывали эту книгу, будучи новичком в мире программирования, то поздравляю с победой! Если вы узнали кое-что и о Unity или другом языке сценариев, то поздравляю вас и с этим! Если вы пришли к нам уже опытным бойцом и рассмотренные темы лишь прочнее закрепились в вашей голове, то — вы угадали — поздравляю! Не существует понятия «незначительного» обучения, независимо от того, как мало вы поняли или сумели сделать. Наслаждайтесь временем, которое потратили на изучение чего-то нового, даже если выучили всего один оператор.

Достигнув конца пути, важно оглянуться назад и вспомнить, какие навыки вы приобрели в процессе. Как и в случае со всем учебным контентом, у вас всегда будет что узнать и изучить, поэтому в данной главе мы закрепим следующие темы и рассмотрим, к каким ресурсам можно обратиться в новом приключении:

- основы программирования;
- применение C# на практике;
- объектно-ориентированное программирование и многое другое;
- подход к проектам Unity;
- сертификаты Unity;
- следующие шаги и дальнейшее обучение.

Верхушка айсберга

Несмотря на то что на протяжении всей книги мы немало поработали с переменными, типами, методами и классами, в C# есть и другие области, которые мы не затронули. Изучение нового навыка — не простое чтение книги без контекста, это кирпичики информации,

уложенные на фундамент имеющихся знаний с помощью цементного раствора практики.

Ниже приведено несколько концепций, которые вам будет полезно изучить по мере продвижения в программировании на C#, независимо от того, работаете вы конкретно с Unity или нет:

- необязательные и динамические переменные;
- подходы к отладке;
- параллельное программирование;
- сетевые интерфейсы и RESTful API;
- рекурсия и отражение;
- выражения LINQ;
- паттерны проектирования.

Возвращаясь к коду, который мы писали на протяжении книги, думайте не только о том, чего мы достигли, но и о том, как взаимодействуют части проекта. Наш код модульный, то есть действия и логика каждого модуля самодостаточны. Наш код гибкий, что позволяет легко улучшать и обновлять его. Наш код чистый, благодаря чему другие люди и мы сами можем в будущем читать и понимать его.

Важно понимать, что усвоение основных концепций требует времени. Не всегда все получается с первой попытки, и моменты озарения не всегда приходят тогда, когда вы этого ожидаете. Главное — продолжать изучать новое, но не забывать повторять фундаментальные темы.

Воспользуемся этим советом и вернемся к принципам объектно-ориентированного программирования в следующем разделе.

Повторим принципы объектно-ориентированного программирования

Объектно-ориентированное программирование — обширная область знаний, и овладение ею требует не только изучения, но и времени, потраченного на применение его принципов в реальной разработке

программного обеспечения. В данной книге мы изучили столько всего (и это лишь основы), что может показаться, будто на эту гору лучше даже не пытаться взобраться. Если у вас возникает такое чувство, то вернитесь на шаг назад и повторите основы ООП.

- Классы — это схемы объектов, которые мы создаем в коде:
  - классы могут содержать свойства, методы и события;
  - у классов есть конструкторы, определяющие создание экземпляров класса;
  - создание объектов из схемы класса создает уникальный экземпляр этого класса.
- Классы — это ссылочные типы, а структуры — это типы «значение».
- Классы могут наследоваться от других классов в целях организации совместного использования общего поведения и данных несколькими подклассами.
- У классов есть модификаторы доступа, позволяющие инкапсулировать их данные и поведение.
- Классы могут состоять из других типов классов или структур.
- Полиморфизм позволяет обрабатывать подклассы так же, как их родительский класс:
  - вдобавок полиморфизм дает возможность изменять поведение дочернего класса, не затрагивая родительский.

Приближение к проектам Unity

Хоть Unity и представляет собой трехмерный игровой движок, он тоже работает по определенным принципам, заданным в коде, на котором написан. Работая над игрой, помните, что игровые объекты, компоненты и системы, которые вы видите на экране, — это просто визуальные представления классов и данных. Это не что-то волшебное, а лишь продукт доведения основ программирования, которые вы изучили в книге, до визуального вывода.

Все в Unity является объектом, но это не означает, что все классы C# должны работать в рамках структуры `MonoBehavior`. Вы можете работать

и за пределами игровой механики. Разветвляйте код и определяйте свои данные или поведение в соответствии с потребностями вашего проекта.

Наконец, всегда думайте, как лучше всего разделить код на функциональные части, и не создавайте огромные, раздутые классы из тысячи строк. Код, объединенный неким общим смыслом, должен сам реализовывать свое поведение и храниться в одном месте. В результате мы создаем отдельные классы `MonoBehavior` и присоединяем их к `GameObject`, на работу которых они влияют. Я говорил это в начале книги и повторю еще раз: программирование — это скорее мышление и структура контекста, а не запоминание синтаксиса. Продолжайте тренировать мозг, чтобы думать как программист, и скоро начнете смотреть на мир иначе.

Возможности Unity, которые мы не рассмотрели

В главе 6 мы кратко осветили многие основные функции Unity, но возможности этого движка гораздо шире. Более сложные темы нельзя расположить в каком-либо определенном порядке по важности, но если вы продолжите работу с Unity, то вам нужно хотя бы мельком ознакомиться с такими темами, как:

- шейдеры и эффекты;
- скриптируемые объекты;
- сценарии расширения редактора;
- непрограммные UI;
- инструменты ProBuilder и Terrain;
- `PlayerPrefs` и сохранение данных;
- создание скелетов моделей;
- состояния и переходы аниматора.

Кроме того, стоит повторить такие инструменты редактора, как Lightning, Navigation, Particle Effects и Animation.

Следующие шаги

Теперь, когда у вас есть базовое понимание языка C#, можно начать развивать дополнительные навыки и изучать новый синтаксис. Чаще всего это подразумевает использование онлайн-сообществ, обучающих сайтов и видеороликов на YouTube, но могут быть и учебники, как данная книга. Перейти от читателя к активному участнику сообщества разработчиков программного обеспечения может быть трудно, особенно с учетом многообразия обучающих ресурсов, поэтому я покажу вам несколько моих любимых ресурсов по C# и Unity, с которых можно начать.

Ресурсы по языку C#

Когда я пишу игры или приложения на C#, у меня всегда открыта документация Microsoft. Если я не могу найти ответ на какой-либо вопрос или проблему, то захожу на свои любимые сайты сообществ:

- C# Corner: www.c-sharpcorner.com;
- Dot Net Pearls: www.dotnetperls.com;
- Stack Overflow: stackoverflow.com.

Поскольку большинство рассмотренных вопросов о языке C# относятся к Unity, мне больше других нравятся именно ресурсы из следующего подраздела.

Ресурсы по Unity

Лучшие учебные ресурсы Unity делают непосредственно авторы: видеоуроки, статьи и документацию на unity3d.com. Но если вам нужны ответы на какие-либо вопросы от сообщества или решение конкретной проблемы программирования, то посетите следующие сайты:

- Unity Learn: learn.unity.com;
- Unity Answers: answers.unity.com;

- StackOverflow: stackoverflow.com;
- Unify Community wiki: wiki.unity3d.com/index.php;
- Unity Gems: Unitygems.com.

На YouTube тоже есть огромное количество видеоуроков от сообществ; вот моя пятерка:

- Brackeys: www.youtube.com/user/Brackeys;
- quill18creates: www.youtube.com/user/quill18creates;
- Sykoo: www.youtube.com/user/SykooTV/videos;
- Renaissance Coders: www.YouTube.com/channel/UCkUIs-k38aDaImZq2Fgsyjw;
- BurgZerg Arcade: www.youtube.com/user/BurgZergArcade.

В библиотеке Packt также можно найти множество книг и видео по Unity, разработке игр и C#: search.packtpub.com/?query=Unity.

Сертификаты Unity

У Unity есть различные уровни сертификации для программистов и художников. Наличие такого сертификата повысит уровень доверия к вашему резюме и позволит оценить ваш уровень навыков. Сертификаты отлично подойдут, если вы пытаетесь ворваться в игровую индустрию в качестве специалиста-самоучки и не работали в IT ранее. Сертификаты бывают следующих видов:

- сертифицированный партнер;
- сертифицированный пользователь: программирование;
- сертифицированный программист;
- сертифицированный художник;
- сертифицированный эксперт — программист игрового процесса;
- сертифицированный эксперт — технический художник: риггинг и анимация;
- сертифицированный эксперт — технический художник: затенение и эффекты.

У Unity также есть подготовительные курсы, продаваемые как самой компанией, так и через сторонних поставщиков. Эти курсы призваны помочь вам подготовиться к различным сертификациям. Вы можете найти подробную информацию на certification.unity.com.

Никогда не позволяйте сертификации или ее отсутствию определять вашу работу или то, что вы вкладываете в мир. Вашим последним «испытанием героя» будет присоединение к сообществу разработчиков, чтобы начать творить историю.

Испытание героя. Выходим в мир

Последнее задание, которое я предлагаю вам выполнить в этой книге, вероятно, самое сложное, но и самое полезное. Ваша задача — использовать свои знания C# и Unity и создать нечто такое, что вы сможете предложить сообществам разработчиков программного обеспечения или игр. Будь то небольшой прототип игры или полномасштабная мобильная игра, поделитесь своим кодом следующими способами:

- присоединяйтесь к GitHub (github.com);
- внесите свой вклад в Unify Community wiki;
- проявляйте активность на Stack Overflow и Unity Answers;
- зарегистрируйтесь для публикации своих ресурсов в Unity Asset Store (assetstore.unity.com).

Каким бы ни был ваш проект, воплотите его в жизнь.

Подведем итоги

Вы можете подумать, что эти строки знаменуют конец вашего пути в программировании, но не стоит заблуждаться. Учебе нет конца, есть только начало. Мы хотели разобраться в основных элементах программирования, основах языка C# и в том, как запрограммировать поведение объектов в Unity. Если вы добрались до этой страницы, то я уверен, что мы выполнили все поставленные задачи.

Последний совет, которого мне не хватало, когда я только начинал: вы программист, если сами считаете себя им. В сообществе будет много людей, которые скажут вам, что вы любитель, вам не хватает опыта, чтобы считаться «настоящим» программистом, или что вам нужна какая-то мифическая печать о том, что вы профессионал. Это не так. На самом деле вы программист, если мыслите как программист, стремитесь решать проблемы эффективно, стараетесь писать чистый код и любите узнавать что-то новое. Помните об этом, и ваш путь на поприще программирования будет потрясающим.

Ответы на контрольные вопросы

Глава 1. Знакомство со средой

Контрольные вопросы. Работа со сценариями

1. Unity и Visual Studio работают в симбиозе.
2. В Reference Manual.
3. Никакую, поскольку это справочный документ, а не тест.
4. Когда на вкладке Project появляется новый файл с редактируемым именем, что делает имя класса таким же, как имя файла, и предотвращает конфликты имен.

Глава 2. Основные элементы программирования

Контрольные вопросы. Основные элементы C#

1. Хранение данных определенного типа для использования в другом месте сценария C#.
2. В методах хранятся фрагменты кода для быстрого и эффективного использования и доступа.
3. Использовать `MonoBehavior` в качестве родительского класса и прикрепить его к `GameObject`.
4. Для доступа к переменным и методам компонентов или файлов, прикрепленных к различным `GameObject`.

Глава 3. Погружение в переменные, типы и методы

Контрольные вопросы. Переменные и методы

1. Использовать стиль *CamelCase*.
2. Объявить переменную публичной.
3. `public`, `private`, `protected`, а также `internal`.
4. Когда явного преобразования нет.
5. Тип данных, возвращаемый методом, имя метода в круглых скобках и пара фигурных скобок, в которых находится блок кода.
6. Чтобы разрешить передачу данных параметров в блок кода.
7. Метод не вернет никаких данных.
8. Метод `Update()` срабатывает в каждом кадре.

Глава 4. Поток управления и типы коллекций

Контрольные вопросы 1. Операторы if, and и or

1. `True` или `false`.
2. Оператор `NOT` — восклицательный знак (`!`).
3. Оператор `AND` — два амперсанда (`&&`).
4. Оператор `OR` — две прямые черты (`||`).

Контрольные вопросы 2. Все о коллекциях

1. Место, где хранятся данные.
2. Первый элемент в массиве или списке имеет индекс 0, так как оба типа нумеруются с нуля.
3. Нет — когда объявляется массив или список, определяется тип данных, которые будут в нем храниться, поэтому элементы не могут принадлежать к разным типам.

4. После инициализации массив нельзя увеличить динамически, поэтому списки более предпочтительны, поскольку их размер можно изменять в процессе выполнения программы.

Глава 5. Работа с классами, структурами и ООП

Контрольные вопросы. Все об ООП

1. Конструктор.
2. По копии, а не по ссылке, как в случае с классами.
3. Инкапсуляция, наследование, композиция и полиморфизм.
4. Метод `GetComponent`.

Глава 6. Погружение в Unity

Контрольные вопросы. Основные функции Unity

1. Примитивы.
2. Ось `z`.
3. Перетащить `GameObject` в папку Prefabs.
4. Ключевые кадры.

Глава 7. Движение, управление камерой и столкновения

Контрольные вопросы. Управление игроком и физика

1. Тип `Vector3`.
2. Компонент `InputManager`.
3. Компонент `Rigidbody`.
4. Метод `FixedUpdate`.

Глава 8. Программируем механику игры

Контрольные вопросы. Работа с механикой

1. Набор или коллекция именованных констант, принадлежащих одной и той же переменной.
2. С помощью вызова метода `Instantiate()` на существующем префабе.
3. Свойства для чтения (`get`) и для записи (`set`).
4. Метод `OnGUI()`.

Глава 9. Основы ИИ и поведение врагов

Контрольные вопросы. ИИ и навигация

1. Генерируется автоматически исходя из геометрии уровня.
2. Компонент `NavMeshAgent`.
3. Процедурное программирование.
4. «Не повторяйся».

Глава 10. Снова о типах, методах и классах

Контрольные вопросы. Новый уровень!

1. Ключевое слово `Readonly`.
2. Изменить количество параметров метода или типы параметров.
3. Интерфейсы не могут иметь реализаций методов или собственных переменных.
4. Создать псевдоним типа, чтобы разделить конфликтующие пространства имен.

Глава 11. Знакомство со стеками, очередями и HashSet

Контрольные вопросы. Сложные коллекции

1. В стеках.
2. Метод `Peek`.
3. Да.
4. С помощью операции `ExceptWith`.

Глава 12. Обобщения, делегаты и многое другое

Контрольные вопросы. Продвинутый C#

1. У обобщенных классов должен быть определенный тип параметра.
2. Сигнатуры метода и делегата.
3. С помощью оператора `-=`.
4. Ключевое слово `throw`.

Харрисон Ферроне

Изучаем C# через разработку игр на Unity. 5-е издание

Перевел с английского А. Павлов

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*М. Сагалович*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*В. Мостипан*
Корректоры	*Е. Павлович, Н. Терех*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 11.2021.

Наименование: книжная продукция.

Срок годности: не ограничен.

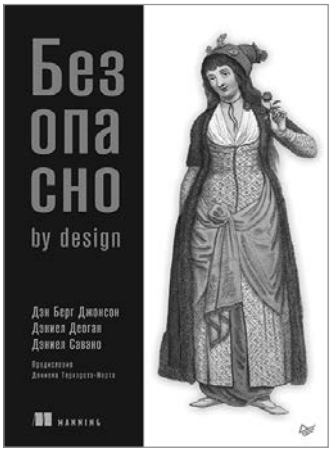
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск,
ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 27.10.21. Формат 70×100/16. Бумага офсетная.
Усл. п. л. 32,250. Тираж 1200. Заказ 0000.

Дэн Берг Джонсон, Дэниел Деоган,
Дэниел Савано

БЕЗОПАСНО BY DESIGN



«Безопасно by design» не похожа на другие книги по безопасности. В ней нет дискуссий на такие классические темы, как переполнение буфера или слабые места в криптографических хэш-функциях. Вместо собственно безопасности она концентрируется на подходах к разработке ПО. Поначалу это может показаться немного странным, но вы поймете, что недостатки безопасности часто вызваны плохим дизайном. Значительного количества уязвимостей можно избежать, используя передовые методы проектирования. Изучение того, как дизайн программного обеспечения соотносится с безопасностью, является целью этой книги. Вы узнаете, почему дизайн важен для безопасности и как его использовать для создания безопасного программного обеспечения.

Мартин Одерски, Лекс Спун, Билл Веннерс

SCALA. ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ. 4-е изд.

«Scala. Профессиональное программирование» — главная книга по Scala, популярному языку для платформы Java, в котором сочетаются концепции объектно-ориентированного и функционального программирования, благодаря чему он превращается в уникальное и мощное средство разработки.

Этот авторитетный труд, написанный создателями Scala, поможет вам пошагово изучить язык и идеи, лежащие в его основе.

Данное четвертое издание полностью обновлено. Добавлен материал об изменениях, появившихся в Scala 2.13, в том числе:

- новая иерархия типов коллекций;
- новые конкретные типы коллекций;
- новые методы, добавленные к коллекциям;
- новые способы определять собственные типы коллекций;
- новые упрощенные представления.

Роберт С. Сикорд

ЭФФЕКТИВНЫЙ С. ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Мир работает на коде, написанном на C, но в большинстве учебных заведений программированию учат на Python или Java. Книга «Эффективный C для профессионалов» восполняет этот пробел и предлагает современный взгляд на C. Здесь рассмотрен C17, а также потенциальные возможности C2x. Издание неизбежно станет классикой, с его помощью вы научитесь писать профессиональные и надежные программы на C, которые лягут в основу устойчивых систем и решат реальные задачи.